JOURNAL OF TELECOMMUNICATIONS AND INFORMATION TECHNOLOGY

4/2002

Cryptographic protocol verification

Special issue edited by Jean Goubault-Larrecq

Foreword

J. Goubault-Larrecq — *Paper* — *1*

Is it possible to decide whether a cryptographic protocol is secure or not?

H. Comon and V. Shmatikov — *Paper* — *5*

CAPSL and MuCAPSL

J. K. Millen and G. Denker — *Paper* — *16*

Process calculi and the verification of security protocols

M. Boreale and D. Gorla — *Paper* — *28*

Asymmetric cryptography and practical security

D. Pointcheval — *Paper* — *41*

Analysis of cryptographic protocols using logics of belief: an overview

D. Monniaux — *Paper* — *57*

CardS4: modal theorem proving on Java smart cards

R. P. Goré and P. Thê Nguyên — *Paper* — *68*

Invited paper

A formal dynamic semantics of Java: an essential ingredient of Java security

M. Debbabi, N. Tawbi, and H. Yahyaoui — *Invited paper* — *81*

ISSN 1509-4553

Foreword

Jean Goubault-Larrecq
Guest Editor

Security is a very old concern, which until rather recently was mostly of interest in military or diplomatic circles, and therefore of rather limited interest to the public at large. The deployment of electronic commerce changed this drastically. This started notably with the advent of computerized communication in the 1970s, and with *smart cards* in the 1980s. Smart cards are really computers on a chip, and need to engage in securized dialogues with cash dispensers, payment terminals or other computer equipment. While this certainly made the concern for security more widespread, notably in the computer and telecommunications industry, it was certainly the hype around Internet and so-called e-commerce – electronic commerce over the World-Wide Web – of the late 1990s that made the notion of security, and in particular of cryptology and cryptographic protocols, a fundamental issue in computers and networks.

One fundamental concern in security is *secrecy*: how do you send a message to a remote friend in such a way that no spy can read it? This led to the art of *cryptography*. Soon other concerns came in, for example *authenticity*: how can your friend be sure that the message he got was indeed created by you, and not forged by a spy? Also, *integrity*: not only should the message originate from you, but it should not have been tampered with or otherwise modified. Also, *freshnes*: the message should come from you, but it should also be recent, and not be a replay of some old message that you sent in ancient times.

Many *encryption* schemes have been devised over the years, from simple permutations of letters to modern algorithms like DES or RSA, or even based on number-theoretic or group-theoretic considerations. Cryptography also expanded its scope, and it was then recognized that not only strong encryption schemes, but also cryptographically secure pseudo-random generators, strong hash functions, and a few other constructs were needed. For a long time, it was believed that designing such strong encryption, random number generation or hashing schemes, was sufficient to ensure secure message exchanges.

Starting from the 1980s, researchers understood that even with perfect encryption schemes, message exchanges were still not necessarily secure. This led to the notion that *cryptographic protocols*, i.e., specified sequences of exchanges of messages, had to be verified and proved before they could be trusted. Although most attacks against cryptographic protocols are relatively trivial, it turned out to be rather difficult to find such attacks, or to be sure that there are none. A case in point is Needham and Schroeder's public-key protocol of 1978 [6], which was discovered flawed 17 years later only, by Gavin Lowe [3].

This issue is dedicated to models and methods for assessing computer security, with a strong emphasis on security of cryptographic protocols. Here is a brief synopsis of the papers in this issue.

Comon-Shmatikov. Hubert Comon and Vitaly Shmatikov survey formal models of cryptographic protocols, comparing the assumptions behind these models, and their respective scopes. These models are the ones that are in use today, and assume perfect cryptographic primitives. Some of them lend themselves to automation. In particular, Hubert Comon and Vitaly Shmatikov discuss decidability issues for cryptographic protocol verification in these settings.

Millen-Denker. One very successful family of models for cryptographic protocols is the so-called Dolev-Yao family of models. A number of automated tools exists that aim at verifying, or finding attacks on cryptographic protocols, based on such models. They include Cathy Meadows' NRL [5], Gavin Lowe and Bill Roscoe's CSP model checker FDR [4] applied to cryptographic protocols, or Jon Millen's CAPSL language, among others. CAPSL is a generic language for describing cryptographic protocols, designed to be interfaced with several verification tools. The architecture and goals of CAPSL are described in this issue by Jon Millen and Grit Denker. They proceed to describe a new extension of CAPSL, μCAPSL, which provides a basis for describing broadcast protocols, such as Diffie-Hellman group authentication – an important topic, although rarely addressed until now.

Boreale-Gorla. Another model for cryptographic protocols stems from process algebra, starting with Martin Abadí and Andrew Gordon's spi-calculus. Principals and intruders are both modeled as processes. A number of classical tools for process analysis – bisimulation, reachability analysis – can be adapted to the cryptographic setting. However this is not trivial, and there are at least two ways that cryptographic properties can be specified and checked on processes. One way is to specify security properties as *indistinguishability*: e.g., a message is secret provided no intruder process can tell the difference between you working with the secret and you working with some other data. Another is *unreachability*: e.g., a message is secret provided you cannot reach a state where the Dolev-Yao intruder knows it. Michele Boreale and Daniele Gorla explore both points of view, and give the basic principles of an automatic cryptographic protocol checker called STA. Note that, surprisingly, the two viewpoints on security are in fact distinct and incomparable.

Pointcheval. David Pointcheval expounds the point of view of the cryptologist, for whom cryptographic primitives are only good up to some probabilistic extent, and assuming complexity-theoretic assumptions. This provides more detailed models than the ones described by Hubert Comon and Vitaly Shmatikov, but are less easily automated. Nonetheless these models provide finer information on the security of cryptographic primitives and protocols. David Pointcheval explains why not every notion of security is relevant here, and proposes a realistic notion of so-called *practical security* for cryptographic protocols.

Monniaux. A popular way of specifying and evaluating the security of cryptographic protocols, starting from a famous paper by Michael Burrows, Martin Abadí, and Roger Needham [1], was the family of cryptographic logics, including BAN and GNY [2]. These logics have been highly criticized since their inception. However, they formalize intuitive reasoning about beliefs of principals in cryptographic protocols. It is widely, but wrongly believed that there is no way we can decide whether a formula in these logics is valid or not. David Monniaux describes the principles behind these logics, and shows why most of these logics are actually decidable, by giving terminating decision algorithms.

Goré-Thê Nguyên. Finally, Rajeev Prabhakar Goré and Phuong Thê Nguyên address the problem of verifying security properties on smart cards. The emphasis here is not so much on security of *data*, in a fixed-code architecture, rather on security in the presence of *mobile code*. The authors survey methods for enforcing security of mobile code, which goes drastically further than plain cryptographic protocols. Then they give a proof-of-principle algorithm for proof search in the modal logic S4, which works on a smart card. This is a *tour de force*, considering how little memory smart card applications have at their disposal. The practical end goal is to enable smart card operating systems to check the conformance of downloaded applications to specific security policies expressed in modal logics in the style of BAN. This will be needed more and more in the future: not just data, but roaming programs must be checked for security.

The domain of computer security and cryptographic protocols is relatively large, and keeps growing. Furthermore, the computer industry is gradually integrating the notion of security in every aspect of computer-related activities. The papers in this issue provide a snapshot of the state of the art in the domain of methods for verifying cryptographic protocols as of 2002, and tries to be as comprehensive as possible. There is no doubt that the domain will flourish in the next years, providing exciting new perspectives.

Acknowledgments

Partially supported by the RNTL project EVA ("Explication et Vérification Automatique de protocoles cryptographiques"), the ACI jeunes chercheurs "Sécurité informatique, protocoles cryptographiques et détection d'intrusions", and the ACI Cryptologie "Psi-Robuste".

References

[1] M. Burrows, M. Abadi, and R. Needham, "A logic of authentication", *Proc. Roy. Soc.*, vol. 426(1871), pp. 233–271, 1989.

[2] L. Gong, R. Needham, and R. Yahalom, "Reasoning about belief in cryptographic protocols", in *IEEE Symp. Res. Secur. Priv.*, 1990, pp. 234–248.

[3] G. Lowe, "An attack on the Needham-Schroeder public-key protocol", *Inform. Proc. Lett.*, vol. 56, no. 3, pp. 131–133, 1995.

[4] G. Lowe and B. Roscoe, "Using CSP to detect errors in the TMN protocol", *IEEE Trans. Softw. Eng.*, vol. 23, no. 10, pp. 659–669, 1997.

[5] C. Meadows, "The NRL protocol analyzer: an overview", *J. Log. Program.*, vol. 26, no. 2, pp. 113–131, 1996.

[6] R. M. Needham and M. D. Schroeder, "Using encryption for authentication in large networks of computers", *Commun. ACM*, vol. 21, no. 12, pp. 993–999, 1978.

Jean Goubault-Larrecq was born in Rouen, France, in 1965. He studied at École Polytechnique, then at the Corps des Mines; he received his Ph.D. degree from École Polytechnique in 1993 and his habilitation in 1997. He worked at the research center of Bull S.A. for ten years, was an invited researcher at the University of Karlsruhe in 1996, worked as an research engineer then project director at Dyade, a common technology transfer venture between Bull and INRIA, from 1996 to 2000. He is currently the full-time professor of computer science at the ENS Cachan, and head of the SECSI (SECurité des Systèmes d'Information) project at INRIA Futurs. His research domains include automated deduction, formal specification, models and methods for cryptographic protocols, intrusion detection, proof theory and modal logics, and algebraic topological methods in computer science.

e-mail: goubault@lsv.ens-cachan.fr
LSV/CNRS UMR 8643
and ENS Cachan ENS Cachan
61, avenue du président-Wilson
F-94235 Cachan Cedex
France

Paper

Is it possible to decide whether a cryptographic protocol is secure or not?

Hubert Comon and Vitaly Shmatikov

Abstract — We consider the so called "cryptographic protocols" whose aim is to ensure some security properties when communication channels are not reliable. Such protocols usually rely on cryptographic primitives. Even if it is assumed that the cryptographic primitives are perfect, the security goals may not be achieved: the protocol itself may have weaknesses which can be exploited by an attacker. We survey recent work on decision techniques for the cryptographic protocol analysis.

Keywords — *cryptographic protocols, decision procedures, logic, security.*

1. Introduction

Security questions are not new. They become increasingly important, however, with the development of the Internet. For example, the classical access control problem, which has been studied in the context of operating systems (e.g. [23, 26]), becomes more complex in a distributed environment where communication channels are not reliable.

How is it possible to secure communications on insecure channels? As we will see, (perfect) cryptographic primitives are a useful tool but security of the primitives does not guarantee security of the protocols. Several protocols had been thought to be secure … until a simple attack was found (see [12] for a survey). Therefore, the question of whether a protocol indeed achieves its security goals becomes crucial.

Until recently, most of the research in protocol analysis was devoted to finding attacks on known protocols, but very few works addressed proof techniques for protocol correctness. This was partly due to the absence of adequate formal models for distributed communications in a hostile environment. In the past 5 years or so, there were proposed several formal models for security protocols (a rough description of the models can be found in Section 2). This opened the way for the use of formal methods and formal analysis of protocols. In this survey, we address the problem of effectiveness of such methods. What can we expect? For what class of protocols are there decision algorithms for security questions?

After explaining the communication and protocol models in Section 2, we discuss the attacker model in Section 3. We then survey the techniques: general techniques (which do not necessarily yield decision algorithms) in Section 4, finite state analysis (which is mainly useful for finding attacks, but does not yield correctness proofs) in Section 5, and, finally, decision results are surveyed in the core of the paper, Section 6.

2. Abstract protocol modeling

In the presence of insecure communication channels, an attacker may be able to observe network traffic and/or intercept messages, modify them in transit, and construct fake messages. In this context, securing communication relies on a set of basic functions that we will refer to as *cryptographic primitives*. For example, an encryption primitive can be used to encode messages prior to transmission on an insecure channel in such a way that the original message content (*cleartext*) can only be retrieved by recipients who possess the "right" decryption key. A number of cryptographic primitives have been designed to achieve information security goals such as secrecy, integrity, authentication, etc.

The analysis techniques discussed in this survey assume *perfect cryptography*. This means that cryptographic primitives are considered as black boxes satisfying certain properties, as described in Section 2.1 below. This assumption by itself does not ensure security of the protocols. Even if all cryptographic primitives used by the protocol are perfectly secure, the protocol itself may have weaknesses which can be exploited by an attacker, as described, e.g., in the Clark and Jacob survey [12]. Typically, an attacker can observe and/or participate in some of the protocol sessions and use the knowledge obtained from these sessions when acting as a participant in subsequent sessions, perhaps impersonating some of the agents. We will give examples of this below. This paper considers the following problem: is it possible to decide, assuming perfect cryptography, whether a given protocol is secure or not?

We must, of course, be more precise about what is a protocol and what is meant by "secure." Informally, a *protocol* is a conversation between two or more agents (also called *principals*) that aims to guarantee certain *security properties* even if a malicious party has access to the communication channel. A more formal definition is given in Section 2.3. In Section 2.4, we describe common security properties such as *secrecy* and *authentication*, and give some examples.

2.1. Cryptographic primitives

In this section, we discuss abstract modeling of cryptographic primitives such as encryption and one-way functions. Other primitives such as digital signatures can be modeled in a similar way.

Symmetric encryption. Suppose that Alice and Bob share a secret value K, which is not known to anybody else. The problem of establishing such a shared secret is beyond the scope of this survey – there are many protocols for achieving this, going back to the Diffie-Hellman key exchange protocol [16]. If Alice wants to communicate privately with Bob, she encrypts her messages with the secret K, producing a (symmetrically encrypted) ciphertext, which we will write as $\{m\}_K^{\leftrightarrow}$. As part of the perfect encryption assumption, we assume that the attacker cannot learn *anything* about m from $\{m\}_K^{\leftrightarrow}$ unless he knows K. In particular, an attacker cannot learn anything by comparing ciphertexts.

Moreover, the attacker cannot construct $\{m\}_K^{\leftrightarrow}$ unless he holds K and m. The attacker may be able, however, to obtain $\{m\}_K^{\leftrightarrow}$ from messages sent by other participants and replay it without learning m. Note that, in theory, the attacker can build all possible keys given a particular key length, and try to decrypt the message with every possible key. The perfect encryption assumption is an idealization of the fact that the attacker has only a very low probability of obtaining the cleartext of an encrypted message within a reasonable time.

Public-key encryption. The symmetric encryption scheme is not practical in many situations since every pair of principals willing to communicate must share a secret. This is the motivation for public-key encryption schemes, of which RSA [46] is the best known. In a public-key encryption scheme, every principal has its own key pair, consisting of a public key K (used for encrypting messages), and a private key K^{-1} (used for decryption). Everybody is allowed to learn the public key K, but the private key is known to its owner only. Therefore, any principal can encrypt messages with K, producing ciphertext $\{m\}_K^{\rightarrow}$, but only the principal who knows K^{-1} can decrypt $\{m\}_K^{\rightarrow}$ to retrieve m. The perfect encryption assumption in this case states that, again, it is impossible to learn m from $\{m\}_K^{\rightarrow}$ without knowing K^{-1}.

One-way functions. Suppose an agent wants to send a long file and be sure that the file is not altered during communication. This can be achieved by sending a digest of the file in a secure way (e.g., digitally signed by the sender). The recipient can then check the integrity of the received message by computing its digest and comparing it with the sender's digest. For this purpose, many protocols make use of *one-way functions*, also called digest functions or *hash functions*. The most widely used hash functions are MD5 [45] and SHA-1 [39]. It is assumed that one-way functions cannot be inverted in the sense that it is computationally infeasible to compute m given $h(m)$, or find m' such that $h(m') = h(m)$.

Nonces. To prevent an attacker from recording messages transmitted as part of one protocol session and replaying them in another session, messages often include nonces. A *nonce* is a value used no more than once for the same purpose [21]. We assume that a nonce is a randomly generated value that satisfies the following properties:

Fresh. If two nonces are generated by different principals or at different times, then they are different.

Unpredictable. An agent or the attacker cannot guess the value of a nonce generated by another agent (although it may able to learn it by analyzing protocol messages).

Many protocols only require freshness, in which case nonces can be replaced by *time stamps*, which we will not consider here.

The decision techniques surveyed in this paper assume, unless explicitly stated otherwise, that neither encryption, nor one-way functions satisfy any algebraic properties. If viewed as term constructors, cryptographic operators form a free term algebra. This assumption does not hold for many functions used in cryptographic applications. For example, xor is self-canceling ($\mathtt{xor}(\mathtt{x},\mathtt{xor}(\mathtt{x},\mathtt{y})) = \mathtt{y}$), and basic RSA satisfies $\mathrm{sig}_{\mathrm{pk}(A)}(\{m\}_{\mathrm{pk}(A)}^{\rightarrow}) = m$ where $\mathrm{sig}_{\mathrm{pk}(A)}(x)$ is agent A's public-key signature of x. There is a wide class of encryption schemes and hash functions, however, for which the free algebra assumption is realistic.

To summarize our view of cryptography, we consider cryptographic functions as abstract black boxes satisfying certain properties. In our model, there is no notion of probability or partial data – the attacker either does not know a value, or knows all bits with 100% certainty. Cryptanalysis attacks that rely on probabilistic properties of cryptographic functions are beyond the scope of the methods considered in this survey. In Section 3.3, we briefly mention recent work on more realistic formal models of cryptography.

2.2. Term algebra

In our abstract model, protocol messages are terms constructed out of:

- Plaintext messages m.
- Nonces.
- Pairing of two messages $\langle M_1, M_2\rangle$ (or, more generally, tupling).
- One-way, unary functions applied to messages $h(M)$.
- Encrypted messages constructed from plaintext M and key k. In general, for symmetric encryption we can view k as an arbitrary term, which provides support, e.g., for symmetric *session keys*, i.e., keys which are generated as part of each instance of the protocol. We will distinguish between public-key and symmetric encryption by using two distinct notations $\{M\}_k^{\rightarrow}$ and $\{M\}_k^{\leftrightarrow}$, respectively. Terms are constructed in the same way in both cases, the only difference is decryption: to decrypt $\{M\}_k^{\leftrightarrow}$, it is necessary to know k, whereas to decrypt $\{M\}_k^{\rightarrow}$, it is necessary to know k^{-1}.

2.3. Protocol specification

A protocol is a process parametrized by a (fixed and finite) set of agents who act as participants. Their names are given as distinct variables $(A, B, \dots)$. Protocol specification consists of a finite sequence of rules of the form $A \to B : M$ where message M is syntactically constructed as described in Section 2.2. The intended (informal) meaning is that A sends to B message M on a public, insecure channel. The names which are used in the ith rule of the protocol refer to the names used in previous steps of the protocol, often in a somewhat ambiguous way, which has to be made precise in the formal models. An *instance* of the protocol, also called a *session* is the image of the protocol by a substitution assigning concrete values to all variables.

2.4. Security properties

While there are many properties that a security protocol may aim to guarantee, in this survey we will be concerned mainly with *secrecy* and *authentication*.

Secrecy. There are many definitions of secrecy, and the relationship between them is not clear [1]. For the purposes of this survey we will say that a protocol preserves secrecy of a datum d is the attacker cannot learn the value of d by interacting with the protocol within the framework of the conventional Dolev-Yao model as described in Section 3. The goal of protocol analysis is then to determine if there exists a protocol trace in which the attacker learns the value of d. It is worth observing that this notion of secrecy is not adequate for, e.g., electronic voting, where possible values of the vote are known in advance and the goal of the protocol is to preserve the confidentiality of the association between a voter and his/her chosen value.

Authentication. There are also many definitions of authentication (see, e.g., [30]). In a nutshell, an event e authenticates agent A if e can occur only if a previous message originated from A. The purpose of authentication is to ensure another agent B that he is indeed talking with A.
Both secrecy and authentication are *trace properties*, i.e., their violations can be found by looking at a single execution trace of the protocol. If the protocol process running in parallel with the attacker process is viewed as a state transition system, the protocol analysis problem for trace properties can be stated as a *reachability* problem, i.e., the problem of determining if the state in which the property is violated is reachable from the protocol's initial state.

There exist security protocols designed to achieve other properties such as fairness, anonymity, non-repudiation, no denial of service, among others, but they are beyond the scope of this survey.

2.5. Example

The following protocol is perhaps the most (in)famous one in the literature on formal analysis of security protocols. It's the (simplified) version of the Needham-Schroeder public-key mutual authentication protocol [40]:

$$\begin{array}{lll} 1. & A \to B & : & \{A, N_A\}_{K_B}^{\to} \\ 2. & B \to A & : & \{N_A, N_B\}_{K_A}^{\to} \\ 3. & A \to B & : & \{N_B\}_{K_B}^{\to} \end{array}$$

In the first message agent A (Alice) sends to agent B (Bob) her name together with a nonce N_A, encrypting the pair with Bob's public key K_B. Bob replies by sending back nonce N_A, together with his own nonce N_B, encrypting the pair with Alice's public key K_A. Finally, Alice sends back Bob's nonce encrypted with K_B.

The goal of the protocol is mutual authentication. After completing the protocol, Alice and Bob should be confident that they are talking to each other. More formally, Alice, upon receiving the second message, should be confident that this message was indeed sent by Bob (since only Bob could decrypt Alice's first message and learn the value of N_A). Bob, upon receiving the third message, should be confident that it was Alice who sent message $\{A, N_A\}_{K_B}^{\to}$ in the first step, since nobody but Alice could decrypt Bob's message and learn the value of N_B. A related goal of the protocol is to preserve the secrecy of nonces N_A and N_B.

Gavin Lowe [29] discovered that the protocol fails to achieve secrecy and authentication due to the following (by now very well-known) attack:

1.1. $A \to I \quad : \quad \{A, N_A\}_{K_I}^{\to}$
The attacker, acting as a legitimate participant in the protocol, is contacted by Alice.

1.2. $I \to B \quad : \quad \{A, N_A\}_{K_B}^{\to}$
The attacker starts a new session of the protocol with Bob, pretending to be Alice.

2.2. $B \to (A) \quad : \quad \{N_A, N_B\}_{K_A}^{\to}$
Bob replies to message 1.2 according to the protocol specification (Bob thinks that message 1.2 came from Alice). Message 2.2 is intercepted by the attacker.

2.1. $I \to A \quad : \quad \{N_A, N_B\}_{K_A}^{\to}$
The attacker replies to message 1.1 using the intercepted message 2.2. At this point Alice, who only observed messages 1.1 and 2.1, believes that N_B has been generated by I.

3.1. $A \to I \quad : \quad \{N_B\}_{K_I}^{\to}$
Alice replies to I's message 2.1 according to the protocol specification.

3.2. $I \to B \quad : \quad \{N_B\}_{K_B}^{\to}$
The attacker, again impersonating Alice, sends the expected answer to message 2.2.

The authentication goals fail as follows:

- Upon reception of message 2.1, Alice should be confident that the message has been constructed by I, which is not the case.

- Upon reception of message 3.2, Bob should be confident that A sent the message $\{A,N_A\}^{\rightarrow}_{K_B}$, which is not the case.

In other words, Bob thinks he is talking with Alice, while he is talking with the attacker.
The secrecy goal fails as follows: N_B should be a secret shared by Alice and Bob only, while message 3.1 allows the attacker to learn it.

3. Attacker model

It is typically assumed that the set of principals consists of two disjoint sets: the honest principals and the attackers. The attackers may include dishonest protocol participants. For most protocols, it is sufficient to analyze the security of the protocol against a single attacker that combines the knowledge and abilities of all dishonest principals.
The honest participants are assumed to follow the rules of the protocol as defined in the protocol specification in a mechanistic way. What they do when they receive a message which does not match their expectation is left unspecified. It is assumed that they do not keep track of previously completed sessions and, more generally, that they do not play an active role in detecting or tracing possible attacks.

3.1. Dolev-Yao model

A common attacker model used in formal analysis of security protocols is the so called *Dolev-Yao model*, inspired by [18]. Following the convention, we used the term Dolev-Yao somewhat loosely. Some of the attacker models described below are in fact richer than the original Dolev-Yao model.
We assume that the attacker can eavesdrop on, remove, and arbitrarily schedule messages sent on public communication channels. It can also create new messages from the pieces of messages it already observed and insert them into the channels. The attacker can split unencrypted messages into pieces and decrypt encrypted terms if it knows the correct decryption key. It is assumed that messages contain enough redundancy so that the recipient can always determine if decryption was successful (e.g., when the attacker decrypts an encrypted nonce $\{N\}^{\leftrightarrow}_{K}$ with a key K', he can always tell whether $K = K'$). In the Dolev-Yao model, the attacker has the choice to intercept any message transmitted on a public communication channel and possibly replace it with a message constructed from his a priori knowledge and parts of the messages previously sent by any participant in this or other session of the protocol.
The steps taken by honest participants following the protocol specification and (non-deterministic) actions of the attacker give rise to an abstract model of the protocol as a state transition system (e.g., [34]). The general approach taken in formal analysis of security protocols is to analyze all feasible traces of the state transition system and determine for each trace whether all of the desired security properties are preserved. This task is complicated by the following considerations:

- There can be arbitrarily many sessions (also known as *instances*) of the protocol which can be interleaved in an arbitrary way.
- One agent can participate in arbitrarily many sessions at the same time. The memory of each agent is, therefore, unbounded (as has been mentioned, an agent's memory is limited to uncompleted sessions).
- The attacker can generate an unbounded number of messages.
- Nonces have limited scope: honest principals forget nonces as soon as the corresponding instance of the protocol has completed.

Among the formal models for protocol traces, the most widely used are CSP [22, 29, 47, 48, 50], higher-order logic [41], multiset rewriting [10, 11], and strand spaces [54]. For information about the relation between different models, see [11].

3.2. Spi-calculus

In the spi-calculus [2] the behaviour of honest protocol participants is formalized as a process in a special-purpose process calculus (basically, an extension of π-calculus [36] with cryptographic operations). This process can be replicated any number of times to model several instances of the protocol running concurrently. The attacker can observe and participate in any communication in any possible way. The model, however, also relies on the perfect cryptography assumption.
Protocol security can then be expressed as *observational equivalence* of two systems. In the first system, an arbitrary process A (which models the public network controlled by the attacker) is run concurrently with the process modeling the actual protocol. In the second system, A is run concurrently with a process modeling an idealized specification of the protocol which is secure by design. If the two systems are observationally equivalent in the process-calculus sense (taking into account cryptographic operations, e.g., $\{N\}^{\rightarrow}_{K}$ and $\{N'\}^{\rightarrow}_{K}$ may not be distinguishable by the attacker who does not know K), then the protocol is secure.
For example, secrecy can be modeled by considering an attacker A that outputs a message on a designated channel when it learns the secret. When run concurrently with the ideal version of the protocol, A cannot possibly learn the secret and thus never outputs on the channel. If A cannot learn the secret from the actual protocol, it will not be able to output on the channel when run concurrently with the protocol, and the two systems will be observationally equivalent, i.e.

$$P[\text{secret}] \mid A \approx_{obs} P[\text{any}] \mid A.$$

In addition to the notions of security supported by the Dolev-Yao model, spi-calculus can be used to analyze implicit flows since the attacker may perform comparisons between observed messages and produce output depending on the comparison results. For instance, the processes may test encrypted (unknown) values for equality and perform actions depending on the result of the test.

3.3. Probabilistic models

Recently, attempts have been made to develop analysis techniques for more realistic formal models of cryptography that go beyond the Dolev-Yao abstraction described in Section 3.1. The goal is to replace "black-box" abstractions of cryptographic primitives with probabilistic models. These models include probabilistic polynomial-time process calculus [27, 28] and more traditional (in the cryptographic sense) simulatability-based models [9, 42–44]. No tools have been developed so far for the mechanized analysis of realistic formal security models.

4. General techniques

In general, there is no algorithm which takes a cryptographic protocol as input and always outputs either "yes, the protocol is secure", or "the protocol is insecure and here is an attack." Both secrecy and authentication are undecidable in the Dolev-Yao model, and so are probably all the interesting properties one might want to check [4, 14, 17]. We give more details on the sources of undecidability below.

Despite this limitation, there exist semi-decision techniques which can be automated in various ways. First, observe that it is possible to design an algorithm which will always find an attack (by the Dolev-Yao attacker) in finite time if an attack exists and may not terminate if the protocol is correct. This can be done by simply enumerating all traces of the protocol's state transition system. Then, in each state, it can be decided if secrecy has been violated, as explained in Section 6.1.

Other semi-decision techniques and tools include, but are not limited to, Paulson's inductive method [41], NRL Protocol Analyzer [33], Athena [51], and abstraction-based techniques by Bolignano [6, 7] (this is by no means a comprehensive list).

There are several sources of undecidability. First, the protocol itself can simulate one step of computation for a universal computation model (e.g., a Turing machine): each state of the machine is an agent who, upon reception of a configuration, sends the next configuration to the appropriate state. The attacker only has to bridge two successive sessions forwarding the last message of one session to the appropriate principal as the first message of the next session. That is why decision methods have to either impose a bound on the number of instances as in [4], or restrict manipulation of the messages (e.g., impose a "single reference to previous messages" restriction [14]).

The second source of undecidability is the ability to generate nonces, which may be used, roughly, to simulate arbitrarily many memory locations and therefore encode machines with unbounded memory [17]. Again, if the number of protocol instances is bounded in advance, this cannot occur. In fact, it is sufficient to bound the total number of nonces which are generated in any trace.

Even if it is assumed that there is a bounded number of instances, it is not yet easy to design a decision algorithm since, according to the Dolev-Yao model, the attacker still has an unbounded number of possible choices at any point. In particular, the number of messages that can be created by the attacker is unbounded. An additional restriction bounding the attacker's memory allows development of finite-state decision techniques.

5. Finite-state analysis

Bounding the number of instances and the number of times each cryptographic operation can be applied by the attacker yields finite-state analysis, which terminates. In this case the protocol can be described by a finite state machine and reachability properties such as secrecy and authentication can be expressed formally, e.g., in some temporal logic. This enables the use of finite-state *model checking* tools such as FDR [29, 48], Murφ [37], and Brutus [13].

Lowe [31] gave a syntactic characterization of a class of protocols such that, for every insecure protocol in the class, there is an attack using a bounded number of sessions and a bounded number of applications of cryptographic primitives (therefore, there is a bound on the number of attacker operations and on the size of terms that the attacker may have to construct). For this class, both the attacker's memory and the number of sessions can be bounded without sacrificing completeness. This enables application of model checking. Moreover, the bounds are quite small in practice.

This result can be seen as a decidability result for the class of protocols which satisfy the assumptions in Lowe's paper [31]. Many of these assumptions are among prudent engineering practices for security protocols proposed by Abadi and Needham [3], but it is not realistic to assume that they are satisfied by a particular cryptographic protocol. Following are some of the requirements defining the class:

- The intended recipient of a message should be able to decompose the message into atomic pieces. This means, for example, that he cannot use part of the message as a black box to be included in the reply, as done, e.g., in Kerberos [25].
- Every message must contain (under encryption) the name of the supposed sender.
- Message fields must contain enough redundancy so that it is always possible to determine the type of

the field. Type confusion between keys, names, nonces, etc. should not be possible.

- There are no temporary secrets. The protocol should be secure under the assumption that everything which is sent in the clear is part of the attacker's initial knowledge.

Among Lowe's requirements is also the restriction of protocols to atomic encryption keys which are either nonces, or basic constants. It should not be possible to build new keys out of existing ones. This assumption, however, is too restrictive for the modeling of "real-world" key exchange protocols such as SSL 3.0 [52] where it is typical for the parties to compute symmetric keys as functions of the shared secret material. In fact, reachability is decidable even in the presence of constructed keys assuming the number of protocol instances is bounded (see Section 6.3).

Stoller [53] demonstrated a more general class $\mathscr{C}$ of protocols for which it is possible to derive, from the protocol specification, a theoretical upper bound on the number of cryptographic function applications that have to be made by the attacker. Stoller also gives a decision algorithm for membership in $\mathscr{C}$. The algorithm is complicated due to the lack of syntactic characterization of the protocol class.

6. Decision results for infinite-state analysis

The protocol analysis techniques surveyed in this section assume that there is a bounded number of protocol sessions, but attacker computations are unbounded. In particular, there are no limits on the depth of terms that can be constructed by the attacker. In all of the techniques, the subject of the analysis is an idealized Dolev-Yao model of the honest protocol participants running in parallel with an attacker who controls the public communication channels. This models execution of the protocol in a hostile environment (we will thus use terms "attacker" and "environment" interchangeably). Therefore, every input to the honest processes from the environment can be viewed as constructed by the attacker.

Typically, specifications of protocol participant's roles contain variables. Variables represent data that the participant does not possess prior to starting the protocol and receives from the environment as part of the protocol. For example, after initiating a key exchange with Bob, Alice may receive a term encrypted with her public key. Since Alice does not know the value of the term before receiving it, it will be denoted by a variable in the specification of Alice's role in the protocol.

For instance, in the Needham-Schroeder example from Section 2.5, Bob (i.e., any agent playing the role of Bob), upon reception of message $\{X,Y\}_{\vec{K}_B}$ will send back $\{N_B,X\}_{\vec{K}_Y}$. Here X,Y are variables since, from Bob's viewpoint, they originated from the environment and their values are not known to Bob apriori. X ranges over arbitrary data and Y ranges over principal names (under the assumption that the agents are able to distinguish principal names from other data). In such a formulation, X could be, for instance, a name or a key or a nonce. Some formalisms assume that each piece of data comes annotated with its type, preventing type confusion attacks [12]. In any case, Bob cannot check that X has been generated by the agent whose name is Y.

6.1. Symbolic protocol models

All of the analysis techniques considered in this section have two main components:

Symbolic reduction. The basic idea behind symbolic reduction is to avoid instantiating variables in the protocol specification unless necessary. This is done by defining a symbolic state transition relation which gives rise to the (finite) symbolic state space of the honest protocol participants. Each symbolic state summarizes an infinite number of concrete states that can be obtained by instantiating variables in the symbolic state specification. Protocol correctness conditions are represented by constraints. A typical constraint is the requirement that every input term received by the honest participants from the environment must be derivable from the environment's initial knowledge combined with the terms sent by the participants on public channels up to that point.

Knowledge analysis. Each technique defines a deduction system for determining whether a particular term can be derived from a given set of terms. Obviously, the deduction system depends on the chosen attacker model. In the Dolev-Yao model, even though the set of terms that can be constructed by the attacker from a given finite initial knowledge is infinite, it is possible to effectively compute a finite tree automaton which accepts this set of terms. This is also true if the initial knowledge of the attacker is a regular set of terms [4, 20, 38]. We will use notation $\mathscr{F}(T)$ for the infinite set of terms that can be derived by the attacker from a particular set T of ground terms.

The protocol analysis problem is then reduced to deciding whether the attacker can instantiate a protocol trace that violates one of the protocol correctness conditions, i.e., if there exists an instantiation of variables computable by the attacker that satisfies the constraints implied by the faulty trace.

6.2. Constructed versus atomic keys

In the simplest Dolev-Yao-style model for symmetric encryption, it is assumed that all symmetric keys are atomic – either constants, or variables that can be instantiated only to constants. This simplifies knowledge analysis, since the set of terms $\mathscr{F}(T)$ that can be derived by the Dolev-Yao attacker from a given term set T is equal to $\mathtt{synth}(\mathtt{analz}(T))$ where $\mathtt{synth}$ and $\mathtt{analz}$ are Paulson's synthesis and analysis closures of term sets. Roughly, $\mathtt{analz}(T)$ is the set of all terms that can be obtained by breaking up and decrypting terms in T, and $\mathtt{synth}(T)$ is all terms that can be obtained by combining, encrypting, and hashing terms in T. With atomic keys, analysis of a term is linear in the depth of the term's structure.

To analyze "real-world" protocols, it is often necessary to extend the model with constructed symmetric keys. In a typical key exchange scenario, two parties exchange a secret, then each derives the shared symmetric key by hashing parts of the shared secret together with nonces and other data. An example of this is master key computation in the SSL 3.0 handshake protocol [52].

6.3. Symbolic decision techniques

In this section, we describe several symbolic decision techniques for security protocols and the assumptions they make about protocols. Unless stated otherwise, all of the methods assume a bounded number of protocol instances but impose no bounds on the attacker's knowledge set $\mathscr{F}$. All results described below hold for the scenarios in which a principal is involved in several parallel sessions. Though only [20] explicitly considers an infinite initial knowledge of the attacker, most of the results described below also apply in this case.

Huima. The origin of symbolic protocol analysis can be traced to the seminal work of Dolev and Yao [18] which applied to a very restricted class of protocols. Huima's paper [24] was the first to present a decision technique for secrecy in cryptographic protocols without a bound on attacker operations. The class of protocols considered in [24] is very general. Protocols are defined using an ad-hoc process algebra formalism, somewhat similar to untyped spi-calculus. Both symmetric and public-key encryption are supported, and constructed keys are allowed.

A standard term rewrite system is defined, representing the attacker's ability to manipulate terms by splitting, decrypting with a known key, encrypting, etc. User-defined symbols are given "semantics" by instantiating one of the pre-defined relation templates. For example, after declaring symbols e and d, the user can declare $P_{\text{symenc}}(e,d)$, meaning that for any terms t_1 and t_2, $d(t_1,e(t_1,t_2)) \rightarrow t_2$. While the templates support explicit decryption operators (and, therefore, a limited equational theory associated with the term algebra), there is no support for commutative and associative operators. See also [38].

For each protocol participant, its local state is defined as $\langle p,Y,c\rangle$ where p is the process representing the corresponding protocol role, Y is a partial variable instantiation function from variable names to terms, and c is a counter used to keep track of fresh values. A symbolic state of the entire protocol is defined as a triple $\langle \mathscr{L},\mathscr{M},\mathscr{C}\rangle$ where $\mathscr{L}$ is a function from participant names to their local states, $\mathscr{M}$ is the set of terms known to the environment (attacker), and $\mathscr{C}$ is a list of constraints that must be satisfiable in order for the state to be reachable. Each constraint has one of the following forms: $Eq(t,t')$, $Ineq(t,t')$, or $InClos(t,M)$ where terms t,t' and term set M may involve variable names. An *InClos* constraint represents the requirement that ground instances of term t must be derivable, using the rewrite system, from the ground instances of terms in M. Such a constraint is satisfiable *iff* there exists a substitution σ such that $\sigma t \in \mathscr{F}(\sigma M)$. (We write σt for the term t in which all variables are replaced according to σ.)

Symbolic reduction is handled by defining a transition relation for symbolic global states that generates a finite symbolic state space with associated constraints (e.g., if a participant receives term x, then $InClos(x,M)$ is added to the constraint list, because the state can only be reached if the environment is capable of generating x). Protocol correctness conditions are also formulated as constraints (e.g., secrecy of term t can be expressed as $\neg InClos(t,M)$), and the two constraint lists are merged. Finally, each terminal symbolic state is transformed in a certain way in order to decide whether there exists a instantiation of variables that satisfies all constraints simultaneously. Note that deciding the existence of an instantiation that satisfies an $InClos(t,M)$ constraint requires deciding the knowledge analysis problem as explained in Section 6.1.

The paper contains no details of the algorithm used to decide the constraint satisfaction problem apart from the list of high-level properties that are supposed to guarantee termination, and the claim that the method is sound and complete.

Amadio-Lugiez-Vanackère. Amadio *et al.* [4, 5] use a untyped process algebra formalism similar to the spi-calculus [2] for specifying protocols. Only symmetric-key encryption with atomic keys is considered. Variables in key positions are handled by brute-force enumeration of all possible substitutions. The decision algorithm is proved NP-hard.

In this approach, symbolic reduction is combined with knowledge analysis. A symbolic state of the protocol is a triple (P,T,E) where P is the state of the process representing the honest participants, T is the finite set of terms representing the attacker's knowledge, and E is an ordered list of constraints $x_1 : T_1, \ldots, x_n : T_n$ such that $T_1 \subseteq \ldots \subseteq T_N$. Each $x_i : T_i$ constraint corresponds to a point in the protocol execution where the accumulated knowledge of the environment consists of terms in set T_i. The values of x_i are the terms which are sent to the honest protocol participants in a trace.

Such constraints are equivalent to Huima's $InClos(x_i,T_i)$ constraints and are satisfiable *iff* there exists a substitution σ such that $\sigma x_i \in \mathtt{synth}(\mathtt{analz}(\sigma T_i))$, i.e., if, after σ instantiates all free variables, x_i is derivable from T_i using operations available to the Dolev-Yao attacker (see Section 3.1). It is worth noting that the characterization of $\mathscr{F}(\sigma T_i)$ as $\mathtt{synth}(\mathtt{analz}(\sigma T_i))$ is only valid if decryption keys are atomic. For each symbolic reduction step, the algorithm checks if the substitution required for the step is compatible with previous substitutions. The algorithm thus decides whether there exists a single substitution that solves all $x_i : T_i$ constraints simultaneously.

As in Huima's approach, to account for the conditional it is necessary to accumulate a separate set of equality constraints as symbolic reduction progresses. Equality constraints are solved by a separate set of rules.

Boreale. Boreale [8] also formalizes abstract models of protocols in a variant of spi-calculus, considering only symmetric-key encryption. Only variables or atomic terms may appear in key positions. The original paper [8] only deals with authentication properties, but the general method can also be used to analyze any reachability property, including secrecy. There is a publicly available analysis tool (STAP), which also handles constructed keys.

The knowledge analysis problem for ground terms is decided using a standard Dolev-Yao deduction system. Protocol execution by honest participants is modeled by a symbolic transition relation that allows messages to contain free variables. As in other methods, exhaustive enumeration of all symbolic traces produces a finite symbolic state space of the protocol.

Protocol analysis is then equivalent to deciding, for each symbolic trace, if it can be solved, i.e., if there exists an instantiation of variables in messages such that in the resulting concrete trace every ground term sent by the environment is derivable from the environment's knowledge at that point using the deduction system. This is the same as deciding the satisfiability of Huima's *InClos* constraints for a particular symbolic trace.

The paper gives a *refinement* decision procedure that works by gradually instantiating variables in the symbolic trace until a *solved form* is obtained in which every sent term is derivable by the environment. The key idea is that any symbolic term can be decomposed into a finite number of irreducible components by splitting pairs and decrypting if the correct key is known to the environment. Therefore, for each message sent by the environment, it is possible to (i) split the sent term into its irreducible components, (ii) split all symbolic terms known to the environment at that point into their components. Since both sets are finite, the symbolic knowledge analysis problem can be decided by checking that the component set of the term is included, modulo unification, in the component set known to the environment.

The refinement process is non-deterministic and may lead to several different solved forms for the same symbolic trace. Completeness is proved by demonstrating that every solution of the symbolic trace is a solution of at least one of the solved forms produced by the algorithm.

Fiore-Abadi. Fiore and Abadi [19] are similar to Amadio *et al.* and Boreale in that they use a variant of untyped spi-calculus with symmetric-key encryption and decryption and a free term algebra. The analysis method supports constructed keys, but completeness is proved only for atomic keys.

The method creates a symbolic computation graph of the honest protocol processes. Paths in the graph represent all possible execution traces of the protocol, and some of them may violate the desired security properties. To determine if there exists a concrete execution trace of the protocol corresponding to the violating path, the paper gives an algorithm for deciding the existence of *realisers* for all symbolic inputs (i.e., message sends) to the process from the environment. A realiser is a substitution for variables such that every resulting ground input term can be derived by the environment from the terms it already knows at that point. Once again, this is equivalent to deciding the satisfiability of all $InClos(t_i, M_i)$ constraints, or finding a substitution σ such that $\sigma t_i \in \mathscr{F}(\sigma M_i)$ for all terms t_i sent by the environment at a point where it knows M_i.

Rusinowitch-Turuani. Rusinowitch and Turuani [49] extend the work by Amadio *et al.* [4] in two directions. First, their model supports public keys as well as constructed symmetric keys. Second, they show that the symbolic knowledge analysis problem is NP-complete for the Dolev-Yao attacker as long as the number of sessions is bounded. The main result of the paper is a polynomial bound on the number of attacker operations that may be needed in order to construct the substitution that realizes the attack. If $t \in \mathscr{F}(M)$, i.e., if the term that must be sent by the environment can be derived from the term set representing the environment's knowledge, then there exists a *normal* derivation of t from M that has a polynomial size. This is similar in spirit to Lowe's "small system" result [31], but with significantly fewer restrictions on the protocol.

The polynomial bound on normal derivations is then used to construct an NP-complete procedure for deciding the protocol insecurity problem. The procedure works by guessing a ground substitution σ for all variables such that the size of the σx term has a polynomial upper bound, then guessing a polynomial sequence of attacker operations $l_1, \dots, l_N$, and finally checking that $\sigma t \in l_N(\dots l_1(\sigma M))$. Such a procedure is obviously impractical for real protocol analysis, but in addition to establishing complexity of the problem, the existence of polynomial normal attacks supports the empirical observation that all Dolev-Yao attacks on cryptographic protocols that have been discovered so far are relatively simple.

Comon-Cortier-Mitchell. Comon *et al.* [14] consider the Dolev-Yao protocol model with support for public keys and constructed symmetric keys. There are two main assumptions. The first one slightly relaxes the finite-sessions requirement by assuming that only a bounded amount of fresh data is generated in all sessions. This means that either there is a finite number of sessions, or else the protocol does not contain any nonce generation steps. This restriction alone is not sufficient for decidability; it is still possible to build a protocol simulating one transition step of a universal computation model. The second restriction states, roughly, that an agent can copy only one piece of any message he receives into any message he sends. This rules out, for instance, simulation of two-stack machines.

The decision technique is based on a reduction to set constraints (e.g. [15]), which in turn are reduced to an automata-theoretic question. The resulting algorithm runs in doubly exponential time.

Millen-Shmatikov. Millen and Shmatikov [35] present a decision technique for reachability properties based on constraint solving. Each honest protocol participant is specified as a *semi-bundle* in the strand space model [54]. A semi-bundle is a strand (i.e., a protocol role) parameterized with variables. The Prolog implementation automatically generates all possible interleavings of the semi-bundles.

Using parameterized strands to represent symbolic traces of the protocol achieves a clean separation between the symbolic reduction problem and the knowledge analysis problem. As in other approaches, deciding the latter is equivalent to solving a system of constraints of the form $t_i : T_i$, where t_i is a term, possibly containing variables, sent by the attacker to the honest processes, and T_i is the set of terms available to the attacker. These constraints are equivalent to Huima's *InClos* constraints, and are satisfiable if $\exists\sigma$ such that $\forall i\ \sigma t_i \in \mathscr{F}(\sigma T_i)$, i.e., every term needed to the stage an attack can be generated by the attacker.

The resulting constraint system is solved by applying a set of constraint reduction rules. The constraint solving procedure is terminating, sound, and complete even in the presence of constructed keys. Unlike the Rusinowitch-Turuani procedure [49], the algorithm is useful in practice and can be applied to the analysis of real protocols.

7. Conclusion

Protocol analysis is a model checking problem [32]: given a model (the protocol) and a property, we want to decide whether the model satisfies the property. As we have seen, however, the model is an infinite state system, and classical model checking techniques can only be used to verify an approximate model. Nevertheless, as with infinite-state model checking techniques, symbolic representation of infinite sets of states (e.g., using constraints) and reasoning about such representations may yield interesting decision results, some of which have been sketched above.

There are still a number of open questions, and more generally, several open areas of research. Let us mention two of them as a conclusion:

- We considered only two particular security properties: secrecy and authentication. While the described techniques may work for other trace properties, there are several security goals which are not trace properties (for instance, anonymity and fairness). There is currently no specification language (such as temporal logic for reactive systems), which is rich enough to express all desired security properties. Design of decision algorithms for such properties is an open problem.

- We assumed that terms and messages are generated by a free algebra. As mentioned above, this is an approximation since most cryptographic primitives satisfy some algebraic properties. Which properties can be supported by the model while preserving decidability is an open question.

Acknowledgements

The authors are grateful to Jean Goubault-Larrecq and Alexandre Boisseau for their comments on an early version of this paper.

This work was partly supported by the RNTL project EVA and the ACI Vernam, for the first author, and DoD MURI "Semantic Consistency in Information Exchange," ONR Grant N00014-97-1-0505, and NSF CCR-9629754 for the second author.

References

[1] M. Abadi, "Security protocols and their properties", in *Foundations of Secure Computation*, F. L. Bauer and R. Steinbrueggen, Eds. NATO Science Series, IOS Press 2000, pp. 39–60 (in *20th Int. Summer School Found. Secure Comput.*, Marktoberdorf, Germany, 1999).

[2] M. Abadi and A. Gordon, "A calculus for cryptographic protocols: the spi calculus", *Inform. Comput.*, vol. 148, no. 1, pp. 1–70, 1999.

[3] M. Abadi and R. Needham, "Prudent engineering practice for cryptographic protocols", *IEEE Trans. Softw. Eng.*, vol. 22, no. 1, pp. 6–15, 1996.

[4] R. Amadio and D. Lugiez, "On the reachability problem in cryptographic protocols", in *Proc. CONCUR, Lecture Notes in Computer Science*. Springer, 2000, vol. 1877, pp. 380–394.

[5] R. Amadio, D. Lugiez, and V. Vanackère, "On the symbolic reduction of processes with cryptographic functions". Tech. Rep. 4147, INRIA, March 2001.

[6] D. Bolignano, "Towards a mechanization of cryptographic protocol verification", in *Proc. 9th Int. Conf. Comput. Aid. Verif. (CAV)*, 1997, pp. 131–142.

[7] D. Bolignano, "Integrating proof-based and model-checking techniques for the formal verification of cryptographic protocols", in *Proc. 10th Int. Conf. Comput. Aid. Verif. (CAV)*, 1998, pp. 77–87.

[8] M. Boreale, "Symbolic trace analysis of cryptographic protocols", in *Proc. 28th International Conference on Automata Languages and Programming (ICALP), Lecture Notes in Computer Science*. Springer, 2001, vol. 2076, pp. 667–681.

[9] R. Canetti, "A unified framework for analyzing security of protocols". IACR Cryptology ePrint Archive 2000/067, Dec. 2000. [Online]. Available: http://eprint.iacr.org.

[10] I. Cervesato, N. Durgin, P. D. Lincoln, J. C. Mitchell, and A. Scedrov, "A meta-notation for protocol analysis", in *Proc. 12th IEEE Comput. Secur. Found. Worksh.*, 1999, pp. 55–69.

[11] I. Cervesato, N. Durgin, P. D. Lincoln, J. C. Mitchell, and A. Scedrov, "Relating strands and multiset rewriting for security protocol analysis", in *Proc. 13th IEEE Comput. Secur. Found. Worksh.*, 2000, pp. 35–51.

[12] J. Clark and J. Jacob, "A survey of authentication protocol literature: version 1.0". Draft paper, 1997. [Online]. Available: http://www-users.cs.york.ac.uk/~jac

[13] E. M. Clarke, S. Jha, and W. Marrero, "Verifying security protocols with Brutus", to appear in *ACM Trans. Softw. Eng. Meth.*

[14] H. Comon, V. Cortier, and J. C. Mitchell, "Tree automata with memory, set constraints and ping pong protocols", in *Proc. 28th International Conference on Automata Languages and Programming (ICALP), Lecture Notes in Computer Science*. Springer, 2001, vol. 2076, pp. 682–693.

[15] W. Charatonik and A. Podelski, "Set constraints with intersection", in *Proc. IEEE Symp. Log. Comput. Sci.*, Warsaw, Poland, 1997.

[16] W. Diffie and M. Hellman, "New directions in cryptography", *IEEE Trans. Inform. Theory*, vol. 22, no. 6, pp. 644–654, 1976.

[17] N. Durgin, P. D. Lincoln, J. C. Mitchell, and A. Scedrov, "Undecidability of bounded security protocols", in *Proc. FLOC Worksh. Form. Meth. Secur. Protoc.*, Trento, Italy, 1999.

[18] D. Dolev and A. Yao, "On the security of public key protocols", *IEEE Trans. Inform. Theory*, vol. 29, no. 2, pp. 198–208, 1983.

[19] M. Fiore and M. Abadi, "Computing symbolic models for verifying cryptographic protocols", in *Proc. 14th IEEE Comput. Secur. Found. Worksh.*, 2001, pp. 160–173.

[20] J. Goubault-Larrecq, "A method for automatic cryptographic protocol verification", in *Fifth International Workshop on Formal Methods for Parallel Programming: Theory and Applications (FMPPTA 2000)*, D. Méry and B. Sanders, Eds., *Lecture Notes in Computer Science*. Springer, 2000, vol. 1800, pp. 977–984.

[21] A. J. Menezes, P. C. van Oorshot, and S. A. Vanstone, *Handbook of Applied Cryptography.* CRC Press, 1997.

[22] C. A. R. Hoare, *Communicating Sequential Processes.* Prentice-Hall, 1985.

[23] M. Harrison, W. L. Ruzzo, and J. Ullman, "Protection in operating systems", *Commun. ACM*, no. 8, pp. 461–471, 1976.

[24] A. Huima, "Efficient infinite-state analysis of security protocols", in *Proc. FLOC Worksh. Form. Meth. Secur. Protoc.*, Trento, Italy, 1999.

[25] J. T. Kohl and B. C. Neuman, "The Kerberos network authentication service (version 5)". Internet Request for Comments RFC-1510, Sept. 1993.

[26] B. Lampson, "Protection", in *Proc. 5th Princeton Conf. Inform. Sci. Syst.*, Princeton, 1971 (reprinted in *ACM Oper. Syst. Rev.*, vol. 8, no. 1, pp. 18–24, 1974).

[27] P. D. Lincoln, J. C. Mitchell, M. Mitchell, and A. Scedrov, "A probabilistic poly-time framework for protocol analysis", in *Proc. 5th ACM Conf. Comput. Commun. Secur.*, 1998, pp. 112–121.

[28] P. D. Lincoln, J. C. Mitchell, M. Mitchell, and A. Scedrov, "Probabilistic polynomial-time equivalence and security analysis", in *Proc. Formal Methods'99, Lecture Notes in Computer Science*. Springer, 1999, vol. 1708, pp. 776–793.

[29] G. Lowe, "Breaking and fixing the Needham-Schroeder public-key protocol using FDR", in *Proc. Tools Algorithms for the Construction and Analysis Systems (TACAS '96)*, T. Margaria and B. Steffen, Eds., *Lecture Notes in Computer Science*. Springer, 1996, vol. 1055, pp. 147–166.

[30] G. Lowe, "A hierarchy of authentication specifications", in *Proc. 10th IEEE Comput. Secur. Found. Worksh.*, 1997, pp. 31–43.

[31] G. Lowe, "Towards a completeness result for model checking of security protocols", *J. Comput. Secur.*, vol. 7, no. 2–3, pp. 89–146, 1999.

[32] B. Bérard, M. Bidoit, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen, *Systems and Software Verification. Model-Checking Techniques and Tools.* Springer, 2001.

[33] C. Meadows, "The NRL protocol analyzer: an overview", *J. Log. Program.*, vol. 26, no. 2, pp. 113–131, 1996.

[34] Z. Manna and A. Pnueli, *Temporal Verification of Reactive Systems.* Springer, 1995.

[35] J. Millen and V. Shmatikov, "Constraint solving for bounded-process cryptographic protocol analysis", in *Proc. 8th ACM Conf. Comput. Commun. Secur.*, 2001.

[36] R. Milner, "Functions as processes", *Math. Struct. Comput. Sci.*, no. 2, pp. 119–141, 1992.

[37] J. C. Mitchell, M. Mitchell, and U. Stern, "Automated analysis of cryptographic protocols using murphi", in *Proc. IEEE Symp. Secur. Priv.*, 1997, pp. 141–153.

[38] D. Monniaux, "Abstracting cryptographic protocols with tree automata", in *Proc. 6th International Static Analysis Symposium (SAS'99)*, *Lecture Notes in Computer Science*. Springer, 1999, vol. 1694.

[39] "Announcement of weakness in the secure hash standard". National Institute of Standards and Technology (NIST), 1994.

[40] R. Needham and M. Schroeder, "Using encryption for authentication in large networks of computers", *Commun. ACM*, vol. 21, no. 12, pp. 993–999, 1978.

[41] L. Paulson, "The inductive approach to verifying cryptographic protocols", *J. Comput. Secur.*, no. 6, pp. 85–128, 1998.

[42] B. Pfitzmann and M. Waidner, "Composition and integrity preservation of secure reactive systems", in *Proc. 7th ACM Conf. Comput. Commun. Secur.*, 2000, pp. 245–254.

[43] B. Pfitzmann and M. Waidner, "A model for asynchronous reactive systems and its application to secure message transmission". IBM Res. Rep. RZ 3304, Dec. 2000.

[44] B. Pfitzmann and M. Waidner, "A model for asynchronous reactive systems and its application to secure message transmission", in *Proc. IEEE Symp. Secur. Priv.*, 2001, pp. 184–200.

[45] R. L. Rivest, *RFC 1321: The MD5 Message-Digest Algorithm*. Internet Engineering Task Force, 1992.

[46] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems", *Commun. ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[47] A. W. Roscoe, "Model-checking CSP", in *A Classical Mind: Essays in Honor of C.A.R. Hoare*. Prentice-Hall. 1994.

[48] A. W. Roscoe, "Modeling and verifying key-exchange protocols using CSP and FDR", in *Proc. 8th IEEE Comput. Secur. Found. Worksh.*, 1995, pp. 98–107.

[49] M. Rusinowitch and M. Turuani, "Protocol insecurity with finite number of sessions is NP-complete", in *Proc. 14th IEEE Comput. Secur. Found. Worksh.*, 2001, pp. 174–190.

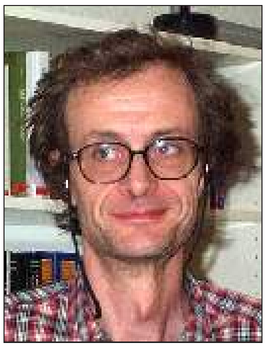
[50] S. Schneider, "Security properties and CSP", in *Proc. IEEE Symp. Secur. Priv.*, 1996, pp. 174–187.

[51] D. Song, "Athena: a new efficient automatic checker for security protocol analysis", in *Proc. 12th Comput. Secur. Found. Worksh.*, 1999, pp. 192–202.

[52] A. Freier, P. Karlton, and P. Kocher, "The SSL protocol. Version 3.0". [Online]. Available: http://home.netscape.com/eng/ssl3/

[53] S. D. Stoller, "A bound on attacks on payment protocols", in *Proc. IEEE Log. Comput. Sci.*, 2001. Available as Tech. Rep. 537, Computer Science Dept., Indiana University, Febr. 2000.

[54] F. J. Thayer Fabrega, J. C. Herzog, and J. D. Guttman, "Strand spaces: proving security protocol correct", *J. Comput. Secur.*, no. 7, pp. 191–230, 1999.



Hubert Comon graduated in mathematics and received his Ph.D. in computer science from INPG, Grenoble in 1988. He is now research director at CNRS and head of the computer science department at ENS Cachan. He worked on rewriting techniques, constraint solving, automated deduction and verification. His main research topics is now on the automatic analysis of cryptographic protocols.
e-mail: comon@lsv.ens-cachan.fr
LSV / École Normale Supérieure de Cachan & CNRS UMR 8643
61, avenue du président-Wilson
94235 Cachan Cedex, France

Vitaly Shmatikov is a computer scientist at SRI International in Menlo Park, California. He is an active researcher in the area of computer security, and has published papers on formal specification and analysis of computer security protocols, modeling techniques, design and implementation of formal analysis tools for computer security, programming language theory and software engineering. He is particularly interested in emerging security properties such as fairness, identity protection, and distributed negotiation. Dr. Shmatikov received his Ph.D. in Computer Science and M.S. in Engineering-Economic Systems from Stanford University, and B.S. in Computer Science and Mathematics from the University of Washington.
e-mail: shmat@csl.sri.com
SRI International
Room EL 239
333 Ravenswood Avenue
Menlo Park, CA 94025, USA

Paper

CAPSL and MuCAPSL

Jonathan K. Millen and Grit Denker

Abstract — Secure communication generally begins with a connection establishment phase in which messages are exchanged by client and server protocol software to generate, share, and use secret data or keys. This message exchange is referred to as an authentication or key distribution cryptographic protocol. CAPSL is a formal language for specifying cryptographic protocols. It is also useful for addressing the correctness of the protocols on an abstract level, rather than the strength of the underlying cryptographic algorithms. We outline the design principles of CAPSL and its integrated specification and analysis environment. Protocols for secure group management are essential in applications that are concerned with confidential authenticated communication among coalition members, authenticated group decisions, or the secure administration of group membership and access control. We will also discuss our progress on designing a new extension of CAPSL for multicast protocols, called MuCAPSL.

Keywords — CAPSL, MuCAPSL, cryptographic protocol specification, cryptographic protocol analysis, secure group communication, multicast.

1. Introduction

In computer networks, cryptography is used to protect private messages and to authenticate the source and content of messages. The range of applications of cryptographic techniques is enormous, including banking, electronic commerce, protection of personal and medical data, trade secrets, and government and military uses. Cryptography supports secure access to World-Wide-Web servers, virtual private networks, and other services over the Internet.

Secure communication generally begins with a connection establishment phase in which messages are exchanged by client and server protocol software to generate, share, and use secret data or keys. This message exchange is referred to as an authentication or key distribution protocol. A few such protocols have been put forward by standards bodies, and others are in common use, such as SSL for securing web page accesses, but new protocols are continually being designed. The perpetual need to design new protocols is due to new technology – cryptographic algorithms, computer hardware, network architectures – and new applications, with special goals such as digital cash, voting or contract signing.

The principal topic of this paper is the design of a formal language, CAPSL, for specifying cryptographic protocols, and how this language plays a role in the analysis of their correctness. CAPSL is useful for addressing the correctness of the protocols on an abstract level, rather than the strength of the underlying cryptographic algorithms. We will also discuss our progress on designing a new extension of CAPSL for multicast protocols, called MuCAPSL.

1.1. Protocol vulnerabilities

Protocols can be analyzed under the assumption of ideal encryption: that is, ciphertext can be decrypted only with the help of the proper key, and ciphertext for a chosen plaintext cannot be generated without the help of the proper key. These assumptions distinguish formal models, with which we are concerned, from computational models, which apply probabilistic and computational complexity reasoning, and from cryptanalysis. We assume that the reader is aware of the distinction between public-key encryption, in which an encrypting key is publicized and the corresponding decryption key is kept private, and symmetric-key encryption, for which the two parties share a common secret key.

Cryptographic protocols are designed to defend against hackers or other adversaries who may have the ability to intercept and modify messages on the network. The attacker may also have a legitimate user identity on the network, or (almost the same thing) may have compromised the secret key of some legitimate user in order to masquerade as that user. Protocol designers also consider attacks in which some secret keys that have been in long-term use, or which were used in the past, may have become compromised due to cryptanalysis, and the protocol should be designed so that such compromised keys cannot be re-introduced or used to compromise new keys. This concept of a worst-case, powerful attacker originated in a paper by Dolev and Yao [15]. Attacks perpetrated by a Dolev-Yao attacker are called *active, message-modification,* or sometimes *man-in-the-middle* attacks.

Here is a well-known example of an authentication protocol, showing how such protocols are expressed in textbooks and papers. We will also show how this protocol is vulnerable to a Dolev-Yao attacker:

$$\begin{aligned}&A \to B: \{A, N_a\}_{PB}\\&B \to A: \{N_a, N_b\}_{PA}\\&A \to B: \{N_b\}_{PB}\end{aligned}$$

This particular protocol is supposed to establish a session between *principals* A and B in such a way that each principal authenticates the identity of the other principal, and they share two session-specific secrets N_a and N_b. This protocol was proposed by Needham and Schroeder in [27]. What is shown here is actually not the entire protocol, but just the handshake that comes after an earlier part in which the necessary public keys are exchanged. We will refer to this protocol as "NSPK."

The bracketed term $\{A, N_a\}_{PB}$ represents the encryption of the concatenation of A and N_a using the public key of B. It is assumed here that A has previously obtained B's public key and that only B has the corresponding secret key, and vice versa for B. The message fields N_a and N_b are *nonces,* meaning that they are *fresh*, in the sense that they have not been used before by the principal that originates them. If they are large enough and randomly generated, they could be used as keys to encrypt subsequent messages.

The security claim of this protocol is that A has given N_a directly only to B, because only B could have decrypted the message in which N_a was introduced. Similarly for B and N_b. The protocol also provides entity authentication, i.e., evidence that the other principal is currently actively participating in the protocol, because it includes acknowledgments from B and A containing the nonces they received.

This abstract message-list style of protocol presentation is often called an "Alice-and-Bob" specification, from the conventional names given to the parties represented by A and B.

There is an active attack on the Needham-Schroeder protocol, found by Lowe [19]. Lowe's attack is illustrated in Fig. 1.

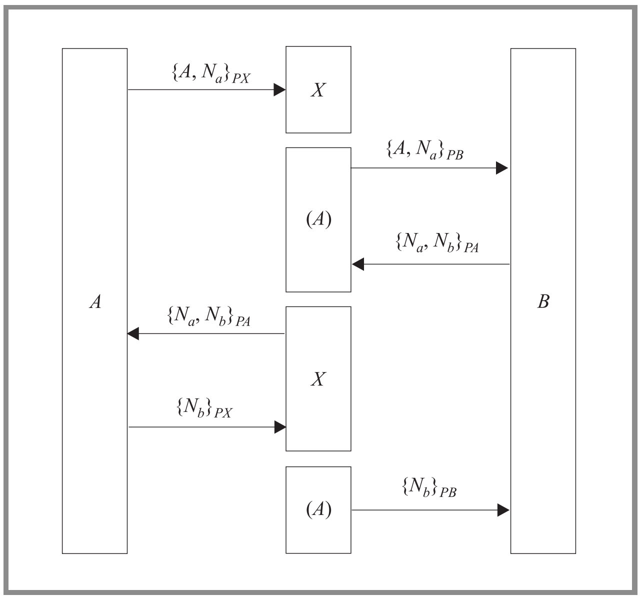


Fig. 1. Lowe's attack.

In this figure, the center column represents the intruder playing two roles. One role is as himself, principal X, responding to A in the left-hand session of the protocol. The intruder is also *masquerading* as A in the right-hand session of the protocol, indicated with (A) in parentheses. There is a security breach in the right-hand session, because B ends up believing he has been talking to A, and that N_b is shared only with A.

1.2. Formal methods

The existence of active attacks led to the development of methods to detect them. Several approaches have been developed, such as specially designed goal-directed state search tools [21, 22] to find attacks, applications of general-purpose specification and verification tools [5, 18, 30] to perform inductive proofs of correctness, specially designed logics of belief [2, 16] to prove authentication properties, and applications of model-checking tools [9, 19, 33], also to search for attacks. These are some of the earliest or most influential papers, and by now the literature is quite extensive.

These tools and their successors have been effective, but it is difficult for analysts other than their developers to apply them. One reason for this difficulty is that the protocols must be respecified for each technique, and it is not easy to transform the published description of the protocol into the required formal system.

Some tool developers began work on translators or compilers that would perform the transformation automatically. The input to any such translator still requires a formally defined language, but it can be made similar to Alice-and-Bob specifications. This is the CAPSL approach. The origins of CAPSL were at the 1996 Isaac Newton Institute Programme on Computer Security, Cryptology, and Coding Theory at Cambridge University.

This approach was also taken by ISL, supporting an application of HOL to an extension of the GNY logic [6]; Casper [20], for the application of FDR using a CSP model-checking approach; and Carlsen's "Standard Notation" [7], which was translated to per-process CKT5 specifications [4].

1.3. The CAPSL approach

The CAPSL language and supporting tools are still under development. This document discusses the design concepts of the language, including the strategy by which CAPSL can be adapted for use by various protocol analysis tools. The basis of this strategy is the use of an intermediate language, CIL, that is close to the state-transition representation used by almost all of these tools. CIL serves two purposes: to help define the semantics of CAPSL, and to act as an interface through which protocols specified in CAPSL can be analyzed using a variety of tools.

CAPSL is parsed and translated to CIL, and there are different translators, called *connectors,* from CIL to whatever form is required for each tool. The translator from CAPSL to CIL can deal with the universal aspects of input language processing, such as parsing, type checking, and unraveling a message-list protocol description into the underlying separate processes. Connectors deal with the semantics and requirements of individual tools. This overall plan is summarized in diagram shown on Fig. 2.

$$\begin{tikzcd} \text{CAPSL} \arrow[r, "\text{Translator}"] & \text{CIL} \arrow[r, "\text{Connector}"] & \text{Analysis tool} \end{tikzcd}$$

Fig. 2. CAPSL translation.

An overview of the CAPSL and CIL environment was given in [11]. The reference report specifying CAPSL is [13], and there is also a web site with CAPSL information [25].

2. CAPSL overview

The acronym "CAPSL" stands for "Common Authentication Protocol Specification Language." A CAPSL specification is made up of three kinds of modules: *TYPESPEC*, *PROTOCOL*, and *ENVIRONMENT* specifications, usually in that order. Typespecs declare cryptographic operators, hash functions, and other operations axiomatically as abstract data types. Specifications for the most popular operators, representing the abstract features of cryptosystems like DES, RSA, and Diffie-Hellman, are included in a standard "prelude" file of typespecs supplied with the CAPSL environment.
Environment specifications are optional; they are used to set up particular network scenarios for the benefit of search tools. We will not discuss environment specifications here.
The core of a protocol specification is a message section containing an Alice-and-Bob specification of the protocol.
An important part of the protocol specification is a statement of its security objectives. There is a *GOALS* section for this purpose, which may include secrecy and authentication statements. Initial assumptions are also specified formally and placed in a section prior to the message list.
Here is a protocol specification for NSPK:

```
PROTOCOL NSPK;
VARIABLES
  A,B: PKUser;
  Na,Nb: Nonce, CRYPTO;
ASSUMPTIONS
  HOLDS A: B;
MESSAGES
  1. A -> B: {A,Na}pk(B);
  2. B -> A: {Na,Nb}pk(A);
  3. A -> B: {Nb}pk(B);
GOALS
  SECRET Na
  SECRET Nb;
  PRECEDES A: B | Na;
  PRECEDES B: A | Nb;
END;
```

Note that declarations may contain property keywords, such as CRYPTO, having some semantic significance. CRYPTO for the variables N_a and N_b means that their values are not guessable (by an attacker). This is significant during analysis, for the attacker model. A variable of type Nonce is assumed by default to have the property FRESH, meaning that values chosen for it have not been used before by the same principal. A nonce is not necessarily CRYPTO, since sequence numbers, i.e., numbers that are increased by one, are FRESH and guessable.

The HOLDS declaration states that the process executing on behalf of *A* has been initialized with the principal *B* chosen as the responder. If the HOLDS assumption is omitted, the CAPSL translator will complain that the sender of the first message does not know the receiver address. By convention, principals always hold themselves.

2.1. Key lookup

Note that the public keys *PA* and *PB* in NSPK have been replaced by function calls `pk(A)` and `pk(B)`. While *A* could have been initialized with *PB*, *B* needs a table lookup, represented by `pk(A)`, to find *PA*, since *B* does not know in advance who will request a connection.
Declaration of key lookup functions is one of the main uses of the abstract type specifications in CAPSL. Such functions are defined for different subtypes of Principal. They embody the kind of long-term key memory a certain kind of principal is assumed to have.
The function `pk` is defined on principals of type PKUser, assumed to have such a table. Principals of type PKUser also have a function `sk` to look up their own corresponding private key. Thus, there is a typespec as follows:

```
TYPESPEC PPK;
TYPES PKUser: Principal;
FUNCTIONS
  sk(PKUser): Pkey, PRIVATE;
  pk(PKUser): Pkey;
VARIABLES
  X: Field;
  P: PKUser;
AXIOMS
  {{X}sk(P)}pk(P) = X;
  {{X}pk(P)}sk(P) = X;
  INVERT {X}pk(P): X | sk(P);
  INVERT {X}sk(P): X | pk(P);
END;
```

The typespec name, in this case PPK, is distinct from the name of the type or types declared in it. Typespec names are used in IMPORTS statements, when necessary. Presently the CAPSL translator does not import typespecs by name; it simply accepts whatever typespecs are provided in its input stream, in order, and requires that a symbol be declared before it is used. Declarations are global, including those of dummy variables used in axioms.
Functions in type specifications are public by default, meaning that both honest principals and the attacker may compute them. However, some functions, like `sk`, deliver long-term secrets such as private keys. These are declared using the keyword "PRIVATE". The first argument of such functions identifies the principal privileged to hold the function value.
There are two kinds of axioms: equational axioms and INVERT axioms. Equational axioms specify the declared functions for the use of theorem provers or term rewriting systems, and also for the use of human readers of the specification, to determine whether the declared abstractions are

suitable as a model for the "real" functions in the protocol. The INVERT axioms are used by the CAPSL translator to determine implementability of a protocol, which will be discussed later.

There are other typespecs in the prelude for principals holding long-term shared symmetric keys, and still others can be added by CAPSL users as needed.

2.2. Goals

Presently two kinds of security goals are supported: SECRET and PRECEDES. A goal `SECRET K;` means that protocol variable K should not be obtainable by the attacker (unless the attacker is acting overtly as one of the legitimate principals in a particular protocol session).

A goal `PRECEDES A: B | K, N;` means that when a principal B reaches the final state of a protocol session, there must be some session of principal A (not necessarily in its final state) that agrees with B on the values of A, B, K and N (or any other variables listed after the $|$, if any). This security goal is meant to represent a fairly general kind of authentication, and it corresponds roughly to formalizations of authentication used by Schneider [34] and Lowe [20].

2.3. Concatenation

A sequence of fields may be concatenated into a single longer field, usually for the purpose of having them encrypted together. Curly brackets { , } and square brackets [,] denote different kinds of concatenation, which are represented by different functions, `cat` and `con` respectively. `cat` is associative and `con` is not.

Both `cat` and `con` are binary. Longer concatenations are parsed under the assumption that right association is intended. Thus, $[a,b,c]$ is parsed as $[a,[b,c]]$.

Associativity of concatenation matters when we try to decompose a concatenation. In particular, the first component of a `cat` term is extracted by `first`, and that of a `con` term by `head`. It is easy to characterize `head` with the axiom `head(con(X,Y)) = X`. But `first(cat(X,Y))` could be either X or `first(X)`, depending on whether or not X is itself a `cat` term.

To deal with this question we differentiate between *atomic* fields, which form the subtype Atom of Field, and those fields that are expressible as a concatenation of smaller fields. The first component of a `cat` concatenation is the first atomic component. Most types are subtypes of Atom.

Another feature of associative concatenation is that a message `A -> B: {C,D}K` can be received by B only if either (1) C is held by B or (2) C is atomic. If C is neither held or atomic, B cannot parse the concatenation from left to right – it won't know where C stops and D begins.

2.4. Other language features

CAPSL has additional syntax to make it more expressive, more concise, and to help resolve ambiguity.

Variables can be introduced to precompute expressions. Suppose, for example, a certain symmetric key K is computed as a hash of other variable values. We can write an equation that looks like an assignment statement:

```
    K = sha({Na,A,Kab});
```

This equation can be placed in the MESSAGES section before a message containing the first use of K. Alternatively, it can be placed in a prior DENOTES section, like a declaration, and it will automatically be used when needed. (This is done with a preprocessing step in the CAPSL translator.) If K is computed in two different ways by different principals (this might happen, for example, when computing a Diffie-Hellman shared key), each DENOTES equation can be labelled by the principal allowed to use it, e.g.,

```
    K = sha({Na,A,Kab}):A;
```

Another useful feature is the % syntax introduced by Lowe in Casper [20]. A message might be computed with an expression by the sender, but handled by the receiver as a black box. The sender of a term $X\%Y$ views it as X but the receiver sees it as Y. Consider the statements:

```
A -> B: {X}pk(C)%Y;
B -> C: Y%{X}pk(C);
```

While A and C understand the message as an encryption, B merely forwards it.

Message sections may also invoke a subprotocol (specified separately as a protocol) using an INCLUDE statement, and make tests using equations to either abort the protocol on a failed test or to choose between IF-THEN branches.

It is one of the characteristics of CAPSL as a specification language that protocol variables receive a value only once, and are not changed after they have been initialized or computed or received. This means that an equation like `K = sk(B);` can be unambiguously identified as either an assignment statement (if K is not defined but B is), a test (if K and B are both defined), or a mistake.

3. The intermediate language CIL

The CAPSL intermediate language (CIL) is designed to make the translation to tool-specific representations as easy as possible. Fortunately, the protocol specifications required for most protocol analysis tools have considerable structural similarity. They generally specify a protocol with state-transition rules for communicating processes. CIL uses multiset term rewriting rules that permit state changes to be presented concisely, and in a way that closely matches the requirements of analysis tools. This approach was influenced by an analysis example using Maude, by Denker, Meseguer, and Talcott, presented at a LICS '98 workshop [14], and by Mitchell's multiset rewriting (MSR) formulation, presented at a Computer Aided Verification workshop in 1998, and also later, in more detail, in [8].

3.1. The MSR model

In the MSR model, the current global state of a network is a multiset containing "facts" representing the current state of some processes engaging in the protocol, and some messages in transit. The network is a multiset simply because it is possible that many copies of the same process state or message might be present simultaneously due to multiple concurrent protocol sessions.
An MSR rule for a state transition in which A handles the exchange

```
B -> A: {K}pk(A);
A -> B: {A,Na}K;
```

might have the abstract form:

$$A_i(A,B), M(\{K\}_{\mathrm{pk}(A)}) \longrightarrow$$
$$(\exists N_a) A_{i+1}(A,B,K,N_a), M(\{A,Na\}_K)\,.$$

Here, "A" is being used both to name a role in the protocol, with states A_i, and as a dummy variable in the rule. The arguments of a state fact are the variables (identified by their positions rather than their names) held by the process. The first argument is always the principal running the process. Message facts are of the form $M(\_)$. The parameter of the message fact holds the content of the message.
In this rule, A in state i decrypts the received message (on the left), adds K to its memory list for the next state $i+1$, generates a nonce N_a, and replies with $\{A\}_K$. The message facts in this rule show only the content of a message, not its source and destination "header." By convention, facts appearing on the left but not on the right are removed from the multiset when the rule is executed.
The existential quantifier in the MSR rule has more than its ordinary logical meaning: it asserts that any value chosen for the variable is a fresh value. This is like a skolemization step when a new constant is chosen to instantiate an existentially quantified variable for proof purposes.
Note that two CAPSL messages have been used to produce a single MSR rule. We could have written two MSR rules for A, one to receive the first message and one to send the second, but then the two rules can be combined. The CAPSL translator actually does this; it processes one message at a time, producing receive-only and send-only rules, and then combines compatible pairs in an optimization step [12].

3.2. CIL vs. MSR

CIL is a variant of MSR. CIL represents state facts in the form

```
state(roleA,i,terms(A,B))
```

and messages in the form

```
msg(B,A,ped(pk(A),K)).
```

Syntactically, these are simply function-term presentations of an abstract syntax tree. Lower-case symbols are node labels, usually function names, and upper-case symbols are variables. Note that the functional representation of `{K}pk(A)` is `ped(pk(A),K)`, and that the CIL version of a message *does* include the source and destination principals.
For the sake of readability, we will continue to use the MSR rather than the CIL form of rules for subsequent explanations.
Another difference between MSR and CIL is that the CIL output of the CAPSL translator includes additional information that is potentially useful for analysis tools, such as a symbol table containing the type signatures of all variable and function names.

3.3. Goals

Presently, goal declarations are translated more or less literally from the CAPSL form to the CIL form, for later use by tool connectors. It is possible to do more, because goals that are security invariants can be converted to MSR and CIL rules that recognize insecure states of the multiset and trigger a "violation" fact. This approach is not difficult to apply manually in a protocol-specific way, but it is is not so easy to set up a general goal translation approach that works for all protocols, even when we restrict the goals to the SECRET and PRECEDES goals in CAPSL. We are investigating general ways to translate goals for future implementation in the translator. In particular, we have found that secrecy goals can be represented with the help of "spell" facts and additional rules as described in [26].

3.4. Implementability

Suppose that a protocol has the messages

```
A -> B: {X}pk(A);
B -> A: X;
```

The transition rule for B could be generated mindlessly as:

$$B_0(B), M(\{X\}_{pk(A)}) \longrightarrow B_1(B,X), M(X)\,.$$

One problem with this rule is that B is actually incapable of decrypting the received message to obtain X. That is, the protocol is *unimplementable.*
The CAPSL translator checks whether a protocol is implementable. In doing so, it deduces what the receiver of a message must do to accept the message, and it also determines whether the sender of a message has the necessary data to construct the message.
Recall that in the MSR notation, a state fact $A_i(\mathbf{y})$ has a sequence $\mathbf{y}$ of terms embodying the memory of the process. In particular, y_1 is the principal for which the process is run. Most of the terms in $\mathbf{y}$ are associated implicitly with protocol variables.
A term t is **computable** from $\mathbf{y}$ by A if

1) $t \in \mathbf{y}$ (sometimes it is convenient to treat $\mathbf{y}$ as a set), or

2) $t = f(\mathbf{x})$ is a functional term whose arguments are computable from $\mathbf{y}$ by A;

and in the second case we check that if f is private, then $x_1 = A$.
Thus, $\mathtt{sk}(B)$ is computable by B from $\{B\}$ but not by $A \neq B$.
A message $M(t)$ can be sent from a state $A_i(\mathbf{y})$ if t is computable by y_1 from $\mathbf{y}$ plus any nonces generated in the state transition rule. If $\mathbf{z}$ is the sequence of those nonces, the next state is $A_{i+1}(\mathbf{y},\mathbf{z})$.
Receivability is more complicated. When a message is encrypted, the receiver must also be able to decrypt it. This is where the INVERT axioms for encryption and other operations come in.
Consider a current state $A_i(\mathbf{y})$ and let $y_1 = A$. A message term m is **receivable** if m is a variable, or m is computable by A, or $m = f(\mathbf{x})$ and for each x_j, either x_j is computable by A or

1) INVERT $f(\mathbf{x}) : x_j \mid \mathbf{w}$ and
2) $\mathbf{w}$ is computable by A, and
3) x_j is receivable.

In each instance of the last case, x_j is *learned* by A, and if the sequence of all learned terms is $\mathbf{z}$, the next state is $A_{i+1}(\mathbf{y},\mathbf{z})$.
The definition above does not allow a message to be receivable if some message fields must be learned before decrypting other fields. For example, the message `A -> B: K, {X}K;` would not be receivable if K was not already held by B. We can handle this message by rewriting it as a pair of messages, with content K and $\{X\}K$ respectively. In fact, this is unnecessary because CAPSL uses a modified receivability algorithm that acts as though such a rewriting had been done. Our algorithm does not presently allow for fields to be sent in reverse order, e.g., `A -> B: {X}K,K;`.

3.5. Connectors

Connectors translate from CIL to some input representation needed by a protocol analysis tool. Connectors have been written for PVS (SRI's verification environment, used for inductive protocol verification [28, 32], Maude [10], Athena [24], and the NRL Protocol Analyzer. The connectors we have written have been in Java, and make use of common connector support classes for parsing CIL and maintaining an internal tree-structured data representation.

4. Secure multicast

Protocols for secure group management are essential in applications that are concerned with confidential authenticated communication among coalition members, authenticated group decisions, or the secure administration of group membership and access control. A variety of new protocols and frameworks have been designed to create multicast groups on a network and support secure group communication (e.g., GDOI [3], GSAKMP [17]. Some existing key exchange protocols for secure communication have been extended to the group setting (e.g., Group Diffie-Hellman GDH [35] and its authenticated form A-GDH [1].
There have been only a few results on the formal analysis of group management protocols (e.g., Pereira and Quisquater analyzed A-GDH [31] and Meadows discovered security flaws in early versions of GDOI [23]. The analysis of group management protocols poses new challenges for formal analysis techniques. New language features and models are necessary to appropriately capture the concepts of such protocols. Moreover, analysis techniques and tools have to be revised and extended.
Multicast CAPSL (MuCAPSL) is an extension of CAPSL affecting all aspects of the environment, from the language and underlying model to analysis techniques and tools. MuCAPSL and its supporting tools are in an early stage of development.

4.1. New MuCAPSL language features

MuCAPSL permits the specification of protocols for secure multicast. The language includes features such as a high-level organization of protocols into suites, a separation of roles for each agent within a protocol, group attributes to capture modifiable persistent state information of group members, and variable-length data structures such as arrays and sequences that are being used as fields in messages or state variables of agents.
In a group setting an agent usually engages in a variety of protocols: to initially set up the group, to distribute new group keys, and to add or delete members. Protocols that conceptually belong together are placed in a *protocol suite*. Within a protocol suite several protocols, typespecs, or environments can be specified. All declarations on the top level of a protocol suite apply to all protocols within the suite. We also refer to the protocols within a suite as *tasks*. A typical suite has the form:

```
SUITE MyGroupMgmt;

TYPESPEC MyGroup;
TYPE MyGroupAgent: GroupMember;
...
END;

PROTOCOL KeyDist;
...
END KeyDist;

PROTOCOL AddMember;
...
End AddMember;


...

END MyGroupMgmt;
```

In the typespec MyGroup we define the type MyGroup-Agent for group members of our example. This type is a subtype of the more general type GroupMember for specifying members of groups.

In a CAPSL typespec we can define principal subtypes and associate immutable or long-term data with a principal by declaring functions. Transient data of principals, associated with a session, is specified via protocol variables. In MuCAPSL, group members store mutable, persistent data in so-called *attributes*. Attributes are shared by different sessions of protocols in a suite and persist between protocol sessions. They are specified in typespecs of group members. There are two functions associated with a group member that jointly serve as a unique identifier: *owner* of type Principal to identify the associated principal (e.g., "Alice"), and *gid* of type Group to identify the associated group (e.g., "Manager" or "Employee"). This way, a principal can be member of several groups. We introduce a new type GroupMember as the default type for group members in the following specification:

```
TYPESPEC GROUPMEMBER;
TYPE GroupMember;
FUNCTIONS
   owner(GroupMember): Principal;
   gid(GroupMember): Group;
END;
```

Note that we also introduced a type Group to capture group identities. In a user-defined group member typespec, the ranges of the functions may be overwritten to subtypes of Principal and Group, respectively.

We will illustrate the use of type specifications for group members using the following simple group attribute structure. Assume a group consisting of N members $M_1, ...M_N$ pairwise sharing long-term symmetric keys. We take advantage of the existing typespec for mutual symmetric key nodes (MSKN) in the CAPSL prelude. It defines a subtype Node of Principal and a function $msk(Node, Node)$ with range Skey. Within a group, members are identified by position number (a natural number), which is a changeable attribute due to members leaving the group or new members joining. At any given time, the leader is the member with position 1. Each member stores a short-term group key K_g, addresses of all group members M_{bs} (defined as an array type, a new parameterized type in the MuCAPSL prelude), and current group size N. Here is the full typespec:

```
TYPESPEC MyGroup;
TYPE MyGroupAgent: GroupMember;
FUNCTIONS owner(MyGroupAgent): Node;
ATTRIBUTES(MyGroupAgent);
   Pos: Nat;
   Kg: Skey, CRYPTO;
   Mbs: Array[Principal];
   N: Nat;
END;
```

Attributes are associated with a group member. The type of group member is specified after the ATTRIBUTES keyword, and should be one of the types (if there are more than one) declared in the typespec. Attributes are like protocol variables because their values may change during execution of protocols, but they are different because a group member state always has values for all of the attributes, though initially, some of them may have an explicit "undefined" value.

We illustrate role-based task specifications of multicast protocols with the help of the key distribution protocol. The leader of the group initiates the key distribution protocol whenever a member has been added to or deleted from the group. We distinguish two main roles in the key distribution: the role of the leader M_1 and the role of other members of the group M_i.

Figure 3 roughly illustrates the message flow of the agent in role M_1. M_1 broadcasts the new group key to the entire group (illustrated in Fig. 3 by the square around the role M_i. A unicast message to a member in role M_i would be depicted by leaving the square out). The member uses a sequence field (denoted by $< ... >$) that includes $N-1$ copies of the new group key, each encrypted with one of the shared keys. The other group members acknowledge the receipt of the group key by each sending a message that contains their position and a nonce encrypted with the group key. The leader collects all responses. We do not specify a specific role from which the leader receives the responses (the lower arrow is not connected to a sending role). This is done intentionally since the leader is not able to reliably check from the addresses who was sending the message since those addresses are easy to fake. The iteration is indicated by a square that contains the condition $i \in 2..N$ expressing that M_1 receives messages of the form $i, \{Nc_i\}K_g$ until it has collected one for each i (possibly out of order).

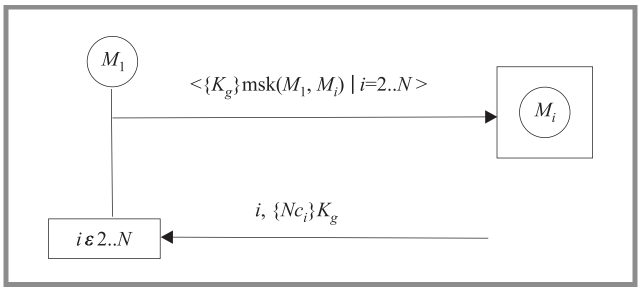


Fig. 3. Key distribution protocol – role M_1.

The separation of roles is also reflected in the protocol specifications. The following shows part of the key distribution protocol, with two roles. The group member playing a role is referred to by the variable implicitly declared in the ROLE statement. The associated principal and group could be derived from this variable via the functions *owner* and *gid*:

```
PROTOCOL KeyDist;

ROLE M1: MyGroupAgent;
VARIABLES i: Nat, FREE;
```

```
            Nc: Array[Nonce];
   ASSUMPTIONS Pos=1;
               owner(M1)=Mbs(1);
   MESSAGES
     New Kg;
     -> : <{Kg}msk(Mbs(1),Mbs(i))|i=2..N>;
     FOR i IN 2..N DO
         <- : i, {Nc(i)}Kg;
     OD;
   END;

   ROLE Mi: MyGroupAgent;
   ASSUMPTIONS Pos > 1;
   MESSAGES ...
   END;

   END KeyDist;
```

Note that the above protocol specification does not refer to the specific senders or receivers of multicast messages. The sender is implicitly the principal playing the role, and the receiver of a multicast message is implicitly the whole group. Recipients of unicast messages can be specified.

The scope of protocol variables that are defined with the attribute FREE is the statement in which the variable is bound to a range of values. For instance, in the statement

```
FOR i IN 2..N DO
  <- : i, {Nc(i)}Kg;
OD;
```

the scope of i is the FOR-loop. Agents do not store the values of free variables in their protocol state or member state.

Attributes of the principal playing the role, such as K_g, can be modified within the protocol. For notational convenience, we refer to attributes in the short form `Kg` instead of the more comprehensive form `Kg(M1)`, since we know that the attributes are associated with `M1`. Note that we need a "New" operator to generate new nonce values. Constructs to express loops or other iterative behavior are necessary to deal with dynamically changing group membership or the need to combine responses from a multicast.

Language features that have not been presented in the current example but have been identified as necessary in the design of MuCAPSL include a syntactical distinction for assignment and tests of group attributes, new built-in cryptographic operators such as a group version of Diffie-Hellman encryption, secret sharing, and threshold encryption.

4.2. New MuCIL features

Hand in hand with the extension of the language goes the extension of the underlying semantic model CIL. In MuCIL new "customized" facts for group member states, protocol states, and multicast messages are introduced. All facts are boolean predicates defined in a functional way that assert either the presence of a group member in the network (*member*(...)) or the existence of an agent in a specific state of a protocol (*state*(...)) or the presence of a multicast message in the network (*mmsg*(...)).

The group member state fact looks as follows:

```
member(owner,gid,terms(attributes))
```

The first parameter refers to the principal that is the member of the group (*owner*), the second parameter identifies the group (*gid*). The third parameter is the list of attributes that is specified for the particular group agent. For the above defined group member type MyGroupAgent, the group member state fact is

```
member(M,G,terms(Pos,Kg,Mbs,N)),
```

with variables of the following types: M : *Principal*, G : *Group*, Pos : *Nat*, Kg : *Skey*, Mbs : *Array(Principal)*, and N : *Nat*.

The protocol state fact of CIL is also extended by a reference to the group identity. Moreover, since there may be several protocols in a suite, the role identifier is composed of the role variable name and the protocol identifier. Thus, a state fact for the group member in role M_1 of the key distribution protocol looks as follows:

```
state(M1,G,roleM1KeyDist,i,terms(...)),
```

with a variable $M1$: *Principal*.

The multicast message fact `mmsg(m)` is simplified compared to CIL since the only parameter it holds is the message content. This is motivated by the fact that sender and receiver addresses in messages do not make a difference from the viewpoint of security analysis since an active attacker can always change addresses. Nevertheless, it may be useful to have this information for purposes such as generating prototypes and the like.

A typical (conditional) rewrite rule in MuCIL contains a member state fact, a protocol state fact and a multicast message fact on both sides. While MSR rules are normally interpreted to delete the left-side facts from the multiset, for MuCIL our convention is to retain message facts implicitly (without repeating them on the right) since they are usually multicast. This is not a real difference, because the attacker can duplicate messages anyway.

The first action and the multicast message of role M_1 in the key distribution protocol are represented by the following MuCIL statement:

```
member(M1,G,terms(1,_,Mbs,N)),
state(M1,G,roleM1KeyDist,0,terms()) ⟶
(∃ Kg) member(M1,G,terms(1,Kg,Mbs,N)),
state(M1,G,roleM1KeyDist,1,terms()),
mmsg(map(lambda(X,ped(msk(Mbs(1),X),Kg)),
         proj(Mbs,2,N)))
```

We do not have a full MSR-style version of MuCIL, but in the above rule we have used some symbols like $\longrightarrow$ for readability that would not appear in the pure functional form of MuCIL. The underscore in the member fact refers to a possibly undefined variable, and the actual MuCIL would put in a new variable identifier of an extended type that permits the undefined value.

The rule states that the position of the agent needs to be 1, and the group size and the array variable `Mbs` that holds the other member's addresses need to be defined. The higher-order map and lambda constructs are typical functional language constructs as found, for example, in ML [29]. $lambda(x,u)$ denotes the function mapping x to u, and $map(f,l)$ returns the list of all elements $f(x)$, where x ranges over the elements of l. The projection operator `proj` and the lambda operator, in conjunction with the map operator, allows us to define an array whose elements are K_g encrypted with each member of the sequence of shared keys.

An instance of a multiset of facts is depicted in Fig. 4.

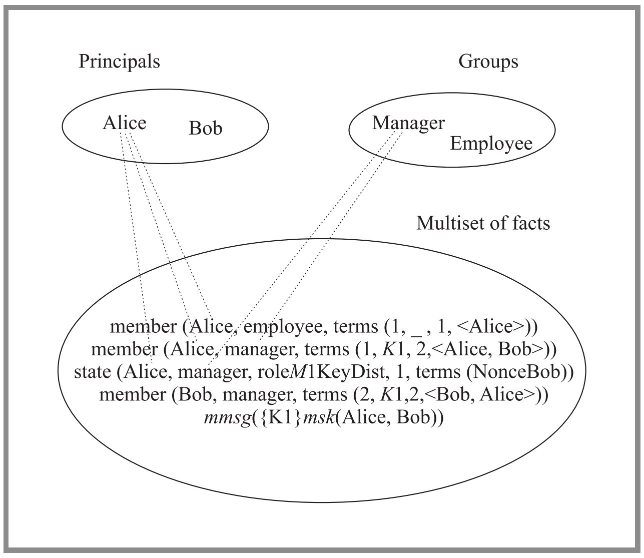

Principals
Groups
Alice
Bob
Manager
Employee
Multiset of facts
member (Alice, employee, terms (1, _ , 1, <Alice>))
member (Alice, manager, terms (1, K1, 2,<Alice, Bob>))
state (Alice, manager, roleM1KeyDist, 1, terms (NonceBob))
member (Bob, manager, terms (2, K1,2,<Bob, Alice>))
mmsg({K1}msk(Alice, Bob))

Fig. 4. A multiset of facts.

4.3. Analysing multicast protocols – preliminary results

We have some preliminary experimentation results in using the Maude model checker for the group Diffie-Hellman protocol (GDH) [35], an extension of the Diffie-Hellman key agreement scheme to an arbitrary group size. The authors of that paper suggest three different protocols that are each optimized with respect to certain protocol complexity such as number of rounds, number of messages, sizes of messages, etc. We analyzed the key distribution protocol GDH.2 as an example since it incorporates unicast messages addressed to a particular agent as well as multicast messages addressed to the group.

The group key in GDH.2 is computed from contributions of each group member. For this purpose, each agent has a nonce N_i. The group key is the exponentiation base a raised to the product of all nonces $\Pi_{i=1..n}N_i$ of group members. The exponentiation base is known to every agent, whereas the individual nonces are secret to the particular group members. In a message exchange, agents communicate partial key values, that keep their secret and still allow other group members to compute a shared group key.

Agents that engage in a GDH protocol are identified by a natural number i. They also keep the current group size n. In GDH.2 we distinguish three roles: M_1, M_i, M_n. M_1 is the role of the group member who initiates the key distribution. This group member is characterized by the identification number 1. The agent in role M_n is the "last" member of the group, the one whose identification number equals the group size. All other members are agents in role M_i.

Figure 5 illustrates the communication between group members for a group of size 4. The agent in role M_1 sends out an array consisting of the exponentiation base a and a^{N1} to its neighbor M_2 (an agent in role M_i). Every agent in role M_i receives such an array, multiplies each array element with its own nonce as well as copies the last array element of the received message in its outgoing message. This way, the length of arrays sent between group members always equals the identification number of the receiving agent. This "upflow" phase of GDH.2 consists of unicast messages. Finally, the last group member receives an array of length n from which it computes the group key by raising the last array element to the power of Nn. Moreover, M_n replies in a multicast to the group with an array of partial key values ("downflow" phase) that include its nonce Nn. The other group member can compute the group key from this multicast message by raising the appropriate array element to the power of their nonce.

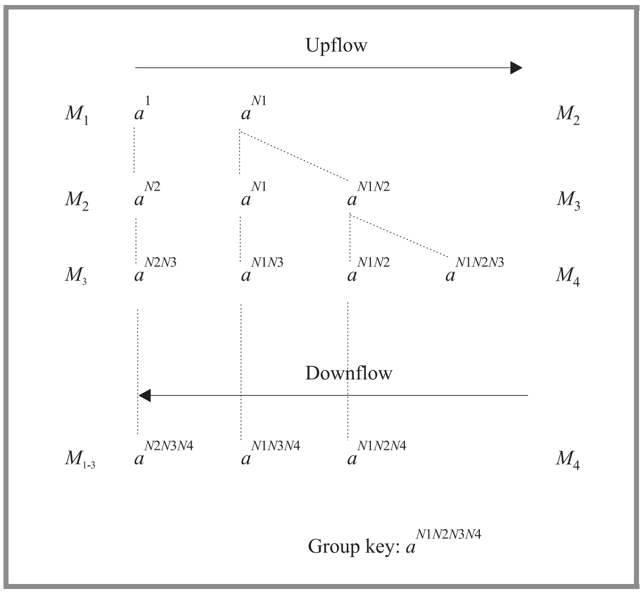


Fig. 5. Overview: group Diffie-Hellman key distribution.

The intent of this protocol is that all group members share a group key. We specified GDH.2 using MuCAPSL and manually translated MuCAPSL into MuCIL. The MuCIL representation was the basis for an analysis using the Maude model checker. In order to deal with MuCIL representations we added arrays, lambda-expressions and undefined values support to the Maude model checker [10, 14]. The Maude model checker has a strategy for state space exploration that computes all possible runs of the protocol for a given initial state. Since most protocols have an infinite state space, Maude's search strategy has a user-definable parameter to limit the state space investigated. The strategy searches for states that violate any of the defined goals. In the case of GDH.2 we declared the following goal of the protocol: all group members agree on the group key after they have finished a run of the key distribution protocol.

We implemented a limited attacker that has the capacity of misdelivery but does not reconstruct messages. In particular, it is possible that messages are delayed, not delivered or delivered several times. We found a state where the leader got the wrong group key. The attack can be generalized to all group members. This is due to ambiguity in the format of GDH.2 messages. The attack is illustrated in Fig. 6 (for a three-member group).

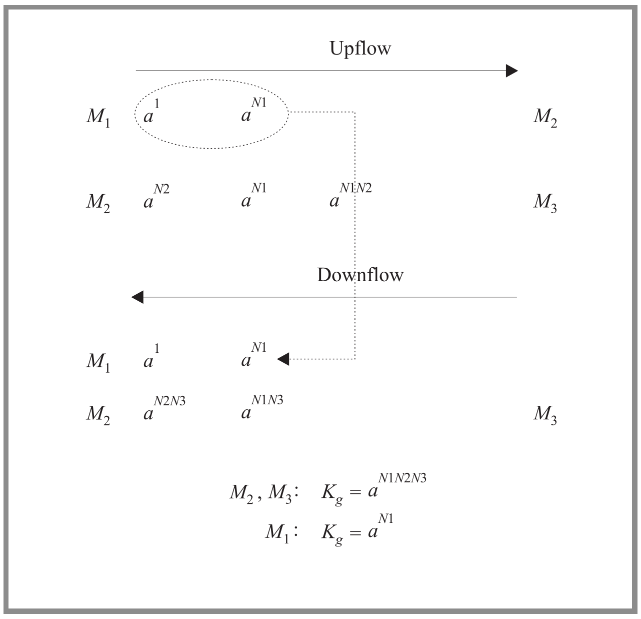

Upflow
M_1 a^1 a^{N1} M_2
M_2 a^{N2} a^{N1} a^{N1N2} M_3
Downflow
M_1 a^1 a^{N1}
M_2 a^{N2N3} a^{N1N3} M_3
M_2, M_3: $K_g = a^{N1N2N3}$
M_1: $K_g = a^{N1}$

Fig. 6. An attack for GDH.2.

GDH.2 as proposed in [35] does not specify whether group members check the content of messages they receive. In fact, in a correct protocol run, a group member cannot decide from its local memory whether a received message has the right content. In the attack the message that member M_1 sent out in the upflow, will be delivered to M_1 in the downflow. Assuming that the agents do not check the content of the message, M_1 computes the group key a^{N_1}. The other members receive and send messages as specified in GDH.2 and therefore compute the group key $a^{N_1 N_2 N_3}$. Group member M_1 has not only been tricked into accepting a wrong group key, but also it uses a group key that is known to the attacker. We would like to stress that GDH.2 was not designed to defeat an active attacker.

5. Conclusion

CAPSL, CIL and the translation between them are designed to address important goals in cryptographic protocol specification for analysis purposes. With a common specification language, it becomes possible to harness the combined power of many tools for protocol analysis in a practical way. The components of the CAPSL environment include transportable software for translation of CAPSL to CIL, and connectors to adapt CIL to the input languages of various analysis tools. This software is still under development.

With CAPSL, one can express protocols in the simplest accepted message-list form. Type specifications in CAPSL and their use for introducing new operators and subtypes bring an expanding class of protocols within reach. CAPSL clarifies what used to be the most awkward aspect of abstract protocol specification, the distinction between short-term session data and the long-term data associated with persistent entities. This was done by applying the general type specification mechanism, together with the novel concepts of private functions and invertibility axioms. In the MSR model, session data is held in state memory.

We have begun to broaden the applicability of CAPSL further with the extension to MuCAPSL for multicast protocols. Protocols that conceptually belong together are grouped into protocol suites. Separation of roles within a protocol was introduced to deal with the concurrent asynchronous operation of protocol processes due to multicast transmission and responses. Message handling within groups is supported by new language constructs for iteration and variable-length data structures.

MuCAPSL is built upon the concepts of CAPSL for type specifications. We added attributes for mutable persistent data. Sequence and array type specifications come along with the new computational operators. On the semantic level, MuCIL has group-member state memory to hold mutable persistent group state attributes. Another extension in MuCIL is the role identifier that uniquely identifies the task and protocol.

The intermediate languages CIL and MuCIL were chosen with an eye toward a clear analysis-level modeling semantics and a universal pattern-matching transition rule style that lends itself both to model checking and inductive proof techniques. We have developed techniques for inductive protocol proof using PVS and model checking using Maude. In our experiments, we have confirmed that CIL output is a good match for the specification needs of these tools. We are currently investigating security goals for multicast protocols, and we are also developing analysis techniques and tools for MuCAPSL.

References

[1] G. Ateniese, M. Steiner, and G. Tsudik, "New multi-party authentication services and key agreement protocols", *IEEE J. Selec. Areas Commun.*, vol. 18, no. 4, pp. 628–639, 2000.

[2] M. Burrows, M. Abadi, and R. Needham, "A logic of authentication", *ACM Trans. Comput. Syst.*, vol. 8, no. 1, pp. 18–36, 1990.

[3] M. Baugher, T. Hardjono, H. Harney, and B. Weis, "The group domain of interpretation". Internet Draft, IETF, 2001. [Online]. Available:
http://www.ietf.org/internet-drafts/draft-ietf-msec-gdoi-01.txt.

[4] P. Bieber, "A logic of communication in a hostile environment", in *Proc. of the Computer Security Foundations Workshop (III)*. IEEE Computer Society Press, 1990, pp. 14–22.

[5] D. Bolignano, "An approach to the formal verification of cryptographic protocols", in *3rd ACM Conference on Computer & Communication Security*. ACM Press, 1996, pp. 106–118.

[6] S. Brackin, "An interface specification language for automatically analyzing cryptographic protocols", in *Symposium on Network and Distributed System Security*. Internet Society, Febr. 1997.

[7] U. Carlsen, "Generating formal cryptographic protocol specifications", in *IEEE Symposium on Research in Security and Privacy*. IEEE Computer Society, 1994, pp. 137–146.

[8] I. Cervesato, N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov, "A meta-notation for protocol analysis", in *12th IEEE Computer Security Foundations Workshop*. IEEE Computer Society, 1999, pp. 55–69.

[9] E. Clarke, S. Jha, and W. Marrero, "Using state space exploration and a natural deduction style message derivation engine to verify security protocols", in *Proc. IFIP Work. Conf. Program. Concepts Meth. (PROCOMET)*, 1998.

[10] G. Denker, "Design of a CIL connector to Maude", in *Workshop on Formal Methods and Computer Security*, H. Veith, N. Heintze, and E. Clarke, Eds. Carnegie Mellon University, July 2000.

[11] G. Denker and J. Millen, "CAPSL integrated protocol environment", in *DARPA Information Survivability Conference (DISCEX 2000)*. IEEE Computer Society, 2000, pp. 207–221.

[12] G. Denker, J. Millen, J. Kuester-Filipe, and A. Grau, "Optimizing protocol rewrite rules of CIL specifications", in *13th IEEE Computer Security Foundations Workshop*. IEEE Computer Society, 2000.

[13] G. Denker, J. Millen, and H. Ruess, "The CAPSL integrated protocol environment". Tech. Rep. SRI-CSL-2000-02, SRI International, 2000.

[14] G. Denker, J. Meseguer, and C. Talcott, "Protocol specification and analysis in Maude", in *Form. Meth. Secur. Protoc.*, 1998, LICS '98 Workshop.

[15] D. Dolev and A. Yao, "On the security of public key protocols", *IEEE Trans. Inform. Theory*, vol. IT-29, pp. 198–208, 1983 (also STAN-CS-81-854, May 1981, Stanford U.).

[16] L. Gong, R. Needham, and R. Yahalom, "Reasoning about belief in cryptographic protocols", in *IEEE Symposium on Research in Security and Privacy*. IEEE Computer Society, 1990, pp. 234–248.

[17] H. Harney, A. Colegrove, E. Harder, U. Meth, and R. Fleischer, "Group secure association key management protocol". Internet Draft, IETF, 2001. [Online]. Available:
http://www.ietf.org/internet-drafts/draft-ietf-msec-gsakmp-sec-00.txt.

[18] R. Kemmerer, "Analyzing encryption protocols using formal verification techniques", *IEEE J. Selec. Areas Commun.*, vol. 7, no. 4, 1989.

[19] G. Lowe, "Breaking and fixing the Needham-Schroeder public-key protocol using FDR", in *Proceedings of TACAS*, *Lecture Notes in Computer Science*. Springer, 1996, vol. 1055, pp. 147–166.

[20] G. Lowe, "Casper: a compiler for the analysis of security protocols", *J. Comput. Secur.*, vol. 6, no. 1, pp. 53–84, 1998.

[21] J. Millen, S. Clark, and S. Freedman, "The Interrogator: protocol security analysis", *IEEE Trans. Softw. Eng.*, vol. SE-13, no. 2, pp. 274–288, 1987.

[22] C. Meadows, "A system for the specification and verification of key management protocols", in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 1991, pp. 182–195.

[23] C. Meadows, "Experiences in the formal analysis of the GDOI protocol". Slides, Dagstuhl Seminar Specification and Analysis of Secure Cryptographic Protocols, 2001. [Online]. Available:
http://www.informatik.uni-freiburg.de/ accorsi/dagstuhl.

[24] J. Millen, "A CAPSL connector to Athena", in *Workshop of Formal Methods and Computer Security*, H. Veith, N. Heintze, and E. Clarke, Eds. CAV, 2000.

[25] J. Millen, "CAPSL web site". 2001. [Online]. Available:
http://www.csl.sri.com/ millen/capsl.

[26] J. Millen and H. Rueß, "Protocol-independent secrecy", in *2000 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2000.

[27] R. Needham and M. Schroeder, "Using encryption for authentication in large networks of computers", *Commun. ACM*, vol. 21, no. 12, pp. 993–998, 1978.

[28] S. Owre, J. Rushby, N. Shankar, and D. Stringer-Calvert, "PVS: an experience report", in *Applied Formal Methods – FM-Trends'98*, D. Hutter, W. Stephan, P. Traverso, and M. Ullman, Eds., *LNCS*. Springer, 1998, vol. 1641, pp. 338–345. [Online]. Available:
http://www.csl.sri.com/papers/fmtrends98.

[29] L. C. Paulson, *ML for the Working Programmer*. 2nd ed. Cambridge University Press, 1996.

[30] L. Paulson, "The inductive approach to verifying cryptographic protocols", *J. Comput. Secur.*, vol. 6, no. 1, pp. 85–128, 1998.

[31] O. Pereira and J. Quisquater, "A security analysis of the cliques protocol suites", in *14th IEEE Computer Security Foundations Workshop*. IEEE Computer Society, 2001, pp. 73–81.

[32] H. Rueß and J. Millen, "Local secrecy for state-based models", in *Form. Meth. Comput. Secur., CAV Worksh.*, Chicago, IL, July 2000.

[33] A. W. Roscoe, "Modelling and verifying key-exchange protocols using CSP and FDR", in *8th IEEE Computer Security Foundations Workshop*. IEEE Computer Society, 1995, pp. 98–107.

[34] S. Schneider, "Verifying authentication protocols in CSP", *IEEE Trans. Softw. Eng.*, vol. 24, no. 9, pp. 741–758, 1998.

[35] M. Steiner, G. Tsudik, and M. Waidner, "Diffie-Hellman key distribution extended to group communication', in *Proc. 3rd ACM Conf. Comput. Commun. Secur.*, 1996.

Jonathan K. Millen is a Senior Computer Scientist in the Computer Science Laboratory at SRI International, working in the area of dependable systems and cryptographic protocol verification. From 1969 to 1997 he worked at the MITRE Corporation. He is co-Editor-in-Chief of the *Journal of Computer Security*, an associate editor of the *ACM Transactions on Information and System Security*, and the founder of the annual *IEEE Computer Security Foundations Workshop*. He holds a Ph.D. in mathematics from Rensselaer Polytechnic Institute, an M.S. from Stanford, and an A.B. from Harvard.
e-mail: millen@csl.sri.com
SRI International
Room EL 233
333 Ravenswood Avenue
Menlo Park, CA 94025, USA

Grit Denker is a Computer Scientist in the Computer Science Laboratory at SRI International in California. From 1995 to 1996 she worked as an assistant professor at the Technical University of Braunschweig in Germany. Her main areas of interest are formal specification and verification of cryptographic security protocols, security for the Semantic Web, and logic-based approaches for distributed system analysis. She holds a Ph.D. in computer science and an M.S. in mathematics from the Technical University of Braunschweig.

e-mail: denker@csl.sri.com
SRI International
Room EL 284
333 Ravenswood Avenue
Menlo Park, CA 94025, USA

Paper

Process calculi and the verification of security protocols

Michele Boreale and Daniele Gorla

Abstract — Recently there has been much interest towards using formal methods in the analysis of security protocols. Some recent approaches take advantage of concepts and techniques from the field of process calculi. Process calculi can be given a formal yet simple semantics, which permits rigorous definitions of such concepts as "attacker", "secrecy" and "authentication". This feature has led to the development of solid reasoning methods and verification techniques, a few of which we outline in this paper.

Keywords — cryptographic protocols, Dolev-Yao model, observational equivalence, process calculi, spi calculus.

1. Introduction

Security protocols have become an essential ingredient of communication infrastructures. When executed in a hostile environment, these protocols may be subject to a number of attacks, that can compromise the security of the data being exchanged over a network. An attacker might typically learn a piece of information which is supposed to remain secret, or it might fool an agent into accepting a compromised key as authentic. Proving a protocol resistant to such attacks is notoriously a difficult task. In the last decade, formal methods have been successfully used to analyse security protocols, sometimes uncovering flaws in protocols that were thought to be correct.

The BAN logic [12] was one of the first, partially successful attempts at using formal methods in the field of security. Later on, finite-state model checking has been extensively used (see e.g. [21, 26]). Some recent developments of formal methods stem from concepts well established in the field of process calculi. In particular, Abadi and Gordon have proposed the *spi-calculus* [3] by elaborating on Milner, Parrow and Walker's π-calculus [24], a process language based on synchronous message passing. The spi-calculus extends the π-calculus with cryptographic primitives, thus allowing the description of security protocols as systems of concurrent processes that can exchange encrypted data. The main advantage of this approach is that process calculi can be given formal yet simple semantics that permit rigorous definitions of such notions as "attacker", "secrecy" and "authentication". Another distinguishing feature of the π-calculus is its reliance on the powerful scoping constructs of the π-calculus to get a clean formalization, at a linguistic level, of such concepts as "nonce", and "newly generated key". In a sense, the spi-calculus improves both the BAN logic, which provides formal reasoning rules but not an operational model, and finite-state methods, which provide a precise operational model but not a convenient basis for formal reasoning. These features have led to the development of solid reasoning techniques and verification methods (e.g. [4, 5, 7, 8, 10]), a few of which we will survey in this paper.

In Section 2 we give a brief overview of the spi-calculus, mainly concentrating on syntax and informal explanation of its operators. Section 3 is devoted to presenting a simplified version of the *Kerberos* protocol [20], which will serve as a running example. While this small protocol is well suited for illustrating the key ideas of the approaches presented here, the reader should be warned that proofs for more sophisticated, in particular multi-session, protocols require a higher degree of ingenuity (see [3, 10]). In Sections 4 and 5 two formal semantics of the spi-calculus are outlined: the first is based on *observational equivalences*, the second is centered around the idea of *trace analysis*. Based on these semantics, rigorous reasoning principles and verification methods are described. Section 6 compares the presented approaches, while Section 7 contains a few concluding remarks and comparison with related work.

2. An outline of the spi-calculus

In this section, we intend to give an informal account of the spi-calculus, by concentrating on syntax and intuitive explanation. The reader is referred to [3, 10] for full technical details.

There are several versions of the spi-calculus. In the rest of this paper, we will consider a variant supporting shared-key cryptography only. This limited language is sufficient to illustrate the key ideas of the approach, while avoiding many technicalities.

Syntax. The syntax of the language is summarized in Table 1. A countable set $\mathcal{N}$ of *names* $a,b\dots,h,k,\dots,x,y,z\dots$ is assumed. Names can be used as variables, communication channels, primitive data or keys: we do not distinguish between these four kinds of objects (notationally, we prefer letters $h,k,\dots$ when we want to stress the use of a name as a key). Messages are built via pairing and shared-key encryption. In particular, $\{M\}_k$ represents the ciphertext obtained by encrypting M under key k, using a shared-key encryption system. An informal explanation of the process operators might be the following:

- $\mathbf{0}$ is the process that does nothing;
- $\tau.P$ does one internal computation step (we do not care precisely what), and then proceeds like P;

- $a(x).P$ waits for a message on channel a and then binds it to variable x within P;
- $\overline{a}\langle M\rangle.P$ sends message M on channel a and then behaves like P;
- $[M = N]P$ behaves like P if the M equals N, otherwise it is stuck;
- $\texttt{case}\, M\, \texttt{of}\, \{y\}_k\, \texttt{in}\, P$ attempts decryption of M using k as a key: if the decryption succeeds, i.e. if $M = \{M'\}_k$ for some M', then M' is bound to variable y within P, otherwise the whole process is stuck;
- $\texttt{pair}\, M\, \texttt{of}\, \langle x,y\rangle\, \texttt{in}\, P$ attempts splitting M; if this is possible, i.e. if M is a pair $\langle M',N'\rangle$, the two components M' and N' are bound, respectively, to variables x and y within P, otherwise the whole process is stuck;
- $(\nu b)P$ creates a new name b which is only known to P;
- $P + Q$ can behave either as P or Q: the choice may be triggered either by the environment or by internal computations of P or Q;
- $P|Q$ is the parallel execution of P and Q;
- $!P$ can be thought of as unboundedly many copies of P running in parallel, i.e. as $P|P|P|\cdots$.

Table 1
Syntax of the calculus

$a,b\ldots,h,k,\ldots,x,y,z\ldots$		*names* $\mathcal{N}$
$M, N ::=$	$a \mid \langle M, N\rangle \mid \{M\}_k$	*messages* $\mathcal{M}$
$P, Q ::=$		*processes* $\mathcal{P}$
	$\mathbf{0}$	(*null*)
	$\mid \tau.P$	(*internal action*)
	$\mid a(x).P$	(*input prefix*)
	$\mid \overline{a}\langle M\rangle.P$	(*output prefix*)
	$\mid [M = N]P$	(*match*)
	$\mid \texttt{case}\, M\, \texttt{of}\, \{y\}_k\, \texttt{in}\, P$	(*decryption*)
	$\mid \texttt{pair}\, M\, \texttt{of}\, \langle x,y\rangle\, \texttt{in}\, P$	(*splitting*)
	$\mid (\nu b)P$	(*restriction*)
	$\mid P + Q$	(*choice*)
	$\mid P\,|\,Q$	(*parallel*)
	$\mid\ !P$	(*replication*)

For the sake of simplicity, we are not considering integer data values present in [3], nor the general form of boolean guard used in [10]. In the definition of this language there are a few implicit assumptions on the underlying shared-key encryption system. We try to make them explicit below:

1) a plaintext M encrypted under a key k can only be decrypted using k; if the attacker does not know k, he/she cannot guess or forge this key (*perfect encryption*);

2) the only way to produce a ciphertext that looks like $\{M\}_k$ is to encrypt M under k;

3) there is enough redundancy in the structure of messages to tell whether a given ciphertext is correctly decrypted with a given key.

The first assumption implies that we can say nothing about attacks that exploit probabilistic or statistical analysis, which may arise in practice, as showed in [28]. In fact, we are concentrating on high-level, logical properties of protocols. The second assumption is an abstraction of the small probability, for real cryptosystems, that different ⟨plaintext, key⟩ pairs collide onto the same ciphertext. The third assumption is in practice implemented by attaching a cryptographic checksum to every plaintext before encryption.
We fix now a few notational shorthands that will be used in the remainder of the paper:

- $a(x).\cdots$ is a binder for x, $\texttt{case}\ \cdot\ \texttt{of}\ \{y\}_k\ \texttt{in}\cdots$ is a binder for y, $\texttt{pair}\ \cdot\ \texttt{of}\ \langle x,y\rangle\ \texttt{in}\ \cdots$ is a binder for x and y and restriction $(\nu b)\cdots$ is a binder for b. We shall also say that x, y and b are *bound* names. Bound names can be renamed to fresh names without affecting the meaning of a process term. We shall always assume that bound names are distinct from each other and from the names that are not bound.
- Names that are not bound are *free*. We use the notation $P(x)$ to emphasize that name x may occur free (i.e. not in the scope of any binder for x) in P and, for any message M, write $P(M)$ to abbreviate $P[^M/_x]$ i.e. P with each free occurrence of x replaced by of M. The set of free names of a process P will be written as $fn(P)$.
- $[M = N, M' = N']$ stands for two consecutive matchings $[M = N][M' = N']$. Similarly, we shall use the shorthands $(\nu a,b)P$ for $(\nu a)(\nu b)P$ and $\texttt{pair}\ M\ \texttt{of}\ \langle x,y,z\rangle\ \texttt{in}\ P$ for $\texttt{pair}\, M\, \texttt{of}\, \langle x,l\rangle\ \texttt{in}\ \texttt{pair}\ l\ \texttt{of}\ \langle y,z\rangle\ \texttt{in}\, P$. The tilde symbol $\tilde{\cdot}$ will be used to denote vectors of objects.

A small example illustrates the use of the calculus for describing cryptographic protocols.

Example. Consider the simple protocol where two principals A and B share a private key k. A wants to send B a datum d encrypted under k, through a public channel c. B accepts any message encrypted with k that is sent along c:

$$A \to B : \{d\}_k \text{ on channel } c.$$

This informal notation can be translated into the spi-calculus process P defined as follows:

$$\begin{aligned} A &\overset{\text{def}}{=} \overline{c}\langle\{d\}_k\rangle.\mathbf{0} \\ B &\overset{\text{def}}{=} c(x).\,\mathtt{case}\,x\,\mathtt{of}\,\{y\}_k\,\mathtt{in}\,F(y) \\ P &\overset{\text{def}}{=} (\nu k)(A|B). \end{aligned}$$

A stops after outputing $\{d\}_k$ on c. B picks up any message from c and then tries to decrypt it using k. If decryption succeeds, the result is bound to variable y within $F(y)$. The latter is some expression describing the subsequent behaviour of B, depending on the result of the decryption, y. The whole protocol P is the parallel composition $A|B$, with the restriction (νk) indicating that the key k is only known to A and B.

On restricted names. The restriction operator plays a crucial role in the spi-calculus. $(\nu k)P$ makes the name k *private* to P. This resembles declarations of local variables in structured programming languages. There is one crucial difference, however: in spi-calculus, a restricted name can be *exported* outside its original scope, while remaining distinct from every name of the recipient. As such, the restriction operator is ideal for modelling those "fresh unguessable quantities" (like random numbers) that are an important ingredient of many cryptographic protocols. The following equation, for instance, explains the creation of a nonce n and its transmission from one principal to another, along a private channel c:

$$(\nu c)\Big(\big((\nu n)\,\overline{c}\langle n\rangle.A\big) \mid c(x).B(x)\Big) = \tau.(\nu c, n)\big(A \mid B(n)\big).$$

The symbol $=$ above can be given a precise meaning in terms of observational semantics, as we shall see in Section 4. Informally, this equation says that the consumption of complementary input and output prefixes $\big(c(x).$ and $\overline{c}\langle n\rangle.\big)$ gives rise to an internal communication (represented by the $\tau.$ prefix) in which n is communicated. This also causes the scope of the restriction (νn) to be extended so as to include B. The scope extension is capture-avoiding, in the sense that n is automatically renamed if it happens to clash with some name in B. This phenomenon is called *scope extrusion* of name n.

A slightly more complicated equation holds when c is a public, rather than private, channel. In this case, the equation also explains the possible interaction of the two principals with the external environment along c.

3. The BAN Kerberos protocol

We shall illustrate the techniques presented in later sections on the version of the Kerberos protocol considered by Burrows, Abadi and Needham in [12]. This section is devoted to an informal presentation of this protocol.

Consider a system where two agents A (the initiator) and B (the responder) share two long-term secret keys, k_{AS} and k_{BS} respectively, with a server S. The protocol is designed to set up a new secret session key k_{AB} between A and B. Informally, the protocol can be described as follows:

$$\begin{aligned} A \longrightarrow S &: A, B \\ S \longrightarrow A &: \big\{T, k_{AB}, B, \{T, k_{AB}, A\}_{k_{BS}}\big\}_{k_{AS}} \\ A \longrightarrow B &: \{T, k_{AB}, A\}_{k_{BS}}, \{A, n_A\}_{k_{AB}} \\ B \longrightarrow A &: \{n_A\}_{k_{AB}}. \end{aligned}$$

In the first message, A starts the protocol by simply communicating to S his intention to establish a new connection with B. In the second message, S generates a fresh key k_{AB} and inserts it into an appropriate certificate, which is sent to A. The certificate uses a timestamp T, meant to assure A and B about the freshness of the message: this is to counter attacks based on replays of old messages. In the third message, A extracts B's part of the certificate $\big(\{\cdots\}_{k_{BS}}\big)$ and forwards it to B, together with some challenge information containing a new nonce n_A. The fourth message is B's response to A's challenge: the presence of n_A is meant to assure A he is really talking to B.

In the next two sections, relying on two different techniques, we shall verify one session configuration of this protocol, under the hypothesis that an old session key $k_{\mathtt{old}}$ between A and B has been compromised. We shall not consider the multi-session case, which requires a more complex analysis. For the sake of simplicity, we shall also suppose that the protocol is always initiated by A and that the responder is always B.

4. Observational equivalences

Following [3], a powerful way of expressing authentication properties of a security protocol P is to require that P is *equivalent* to a process Q that, by definition, exhibits the desired behaviour (e.g., Q never accepts non-authentic messages). Secrecy as well can be expressed via this notion of equivalence. For example, let $P(d)$ be a process in which a secret datum d is exchanged, properly encrypted, along a public channel. A way of asserting that $P(d)$ keeps d secret is requiring that $P(d)$ be equivalent to $P(d')$, for every other d'. An appropriate notion of equivalence is here *may-testing* [3, 9, 14]. Its intuition is precisely that *no* external observer (which in the present setting can be read as "attacker") can notice any difference when, e.g., running in parallel with $P(d')$ or $P(d)$. Formally, we define an observer as a process that is possibly capable of a distinct "success" action ω; the latter is used to signal that the observed process has passed observer's test. If one interprets "passing a test" as "revealing a piece of information", then processes that may pass the same tests may potentially reveal the same information to external observers: as such, they should be considered equivalent from a security point of view. This also accounts for implicit information flow, by which an observer might extract useful information from the overall behaviour of a system.

In the definition below, $R \overset{\omega}{\Longrightarrow}$ means that R can execute zero or more internal computation steps, followed by an ω–action.

Definition 1 (may-testing). Two spi-calculus processes P and Q are *may-testing equivalent*, written $P \simeq Q$, if for every observer O, $P \,|\, O \stackrel{\omega}{\Longrightarrow}$ iff $Q \,|\, O \stackrel{\omega}{\Longrightarrow}$.

A similar intuition is supported by other contextual equivalences, like *barbed equivalence* [25]. While rigorous and intuitive, the definitions of these equivalences suffer from universal quantification over contexts (attackers), that makes equivalence checking very hard. It is then important to devise proof techniques that avoid such quantification.

Results in this direction are well-known for traditional process calculi. For example, both in CCS [14] and in the π-calculus [9], may-testing is easily proven to coincide with *trace equivalence*, which requires that two equivalent processes generate the same sequences of *actions* (I/O events). Similarly, barbed equivalence is proved to coincide with *early bisimulation*. The latter requires that each action of one process be "simulated" by the other, and that the target processes be still bisimilar. In this section we outline a way of obtaining similar results in the case of the spi-calculus; full details can be found in [10]. We then discuss a few resulting reasoning rules and apply them to the Kerberos protocol.

4.1. A labelled transition system for the spi-calculus

In non cryptographic calculi (like the π-calculus) processes and observers share the same knowledge of names. This means, in essence, that the external environment may enable any action that a process is willing to take. This is not true anymore when moving to the spi-calculus. In fact, consider the process P that sends a fresh name b encrypted with a fresh key k and then executes P'. This is written $(\nu\, b,k)\overline{c}\langle\{b\}_k\rangle.P'$. When an observer receives $\{b\}_k$, it does not acquire automatically the knowledge of b, because k is still secret. Thus, if P' is willing to input something at b (say $P' \stackrel{\text{def}}{=} b(x).P''$), the environment cannot satisfy P''s expectations. For this reason, execution traces *à la* π-calculus fail to capture the interactive behaviour of processes.

This discrepancy leads us to make the concept of *environment* explicit, as a record of the knowledge of names and keys that an external observer has acquired about a certain process. More precisely, we model an environment as a mapping σ from a set of variables to a set of messages. Intuitively, an environment is a set of locations named by distinct variables, where an observer (usually an attacker) will store information known. We want now to describe how the environment is modified by the actions performed by the process and how actions that the process can perform are constrained by the environment. To this purpose, we introduce an environment-sensitive labelled transition system (written e.s.–lts in the sequel), whose states are *configurations* $\sigma \triangleright P$, where σ is the current environment and P is a process. Transitions between configurations represent interactions between σ and P, and take the form

$$\sigma \triangleright P \xmapsto[\delta]{\mu} \sigma' \triangleright P',$$

where μ is the action of process P and δ is the complementary *environmental action*. More precisely, μ can be of three forms: an internal action – τ – an input – aM – or an output – $(\nu\widetilde{b})\overline{a}\langle M\rangle$. The latter makes explicit the private names $\widetilde{b}$ that are being extruded. Accordingly, the environmental action δ is a "no-action", an output or an input. Therefore, three kinds of transitions may arise:

1. The process performs an output and the environment an input. As a consequence, the environment's knowledge gets updated. For instance:

$$\sigma \triangleright P \xmapsto[z(x)]{(\nu\widetilde{b})\overline{a}\langle M\rangle} \sigma[{}^{M}\!/\!{}_{x}] \triangleright P',$$

where $\sigma[{}^{M}\!/\!{}_{x}]$ is the update of σ with the new entry $[{}^{M}\!/\!{}_{x}]$, for a fresh variable x. Here, $\widetilde{b}$ is the set of private names the process extrudes. For the transition to take place, channel a must belong to the knowledge of σ, which in this case amounts to saying that $\sigma(z) = a$.

2. The process performs an input and the environment an output. Notice that messages from the environments cannot be arbitrary, but must be built via encryption, decryption, pairing and projection, from the messages recorded in σ, plus some fresh names the environment can create. Thus, a transition might be:

$$\sigma \triangleright P \xmapsto[(\nu\widetilde{b})\overline{z}\langle\zeta\rangle]{aM} \sigma[{}^{\widetilde{b}}\!/\!{}_{\widetilde{b}}] \triangleright P'.$$

Here, $\widetilde{b}$ is the set of new names the environment has just created and added to its knowledge, while ζ is an expression describing how M has been built out of σ and $\widetilde{b}$. This expression uses the variables in the domain of σ. For example, if $\sigma = [{}^{c}\!/\!{}_{x_1}, {}^{k}\!/\!{}_{x_2}, \ldots]$ and $M = \{c\}_k$, then ζ might be $\{x_1\}_{x_2}$, indicating that message M results from encrypting the x_1–entry using the x_2–entry as a key. Again, a must belong to the knowledge of σ, thus $\sigma(z) = a$.

3. The process performs an internal move and the environment does nothing:

$$\sigma \triangleright P \xmapsto[-]{\tau} \sigma \triangleright P'.$$

Having introduced the e.s.-lts, we can define a new equivalence on top of it. The equivalence should only relate configurations that exhibit equivalent environments. Informally, two environments are equivalent if there is no way of telling them apart by performing elementary operations (like projection, decryption, comparison and so on) on their entries. For instance, $\sigma \stackrel{\text{def}}{=} [{}^{a}\!/\!{}_{x}, {}^{b}\!/\!{}_{y}, {}^{\{a\}_k}\!/\!{}_{z}]$ and $\sigma' \stackrel{\text{def}}{=} [{}^{a}\!/\!{}_{x}, {}^{b}\!/\!{}_{y}, {}^{\{b\}_k}\!/\!{}_{z}]$ are equivalent, while $\sigma[{}^{k}\!/\!{}_{w}]$ and $\sigma'[{}^{k}\!/\!{}_{w}]$ are not, because k enables decryption of the z-entry, and then comparing the obtained cleartext with the first two entries yields different results. A formalization of these concepts can be found in [10]; for our purposes, this informal explanation suffices. The taken point of view is that two equivalent configurations should exhibit the *same environmental actions*, no matter what the process actions are. These consideration lead to the definition below. We write $\Longmapsto$ for the reflexive and transitive closure of $\overset{\tau}{\underset{-}{\longmapsto}}$ (i.e., a sequence of zero or more $\overset{\tau}{\underset{-}{\longmapsto}}$ transitions) and, inductively, $\overset{s}{\underset{u}{\Longmapsto}}$ for $\Longmapsto\overset{\mu}{\underset{\delta}{\longmapsto}}\overset{s'}{\underset{u'}{\Longmapsto}}$ when $s = \mu \cdot s'$ and $u = \delta \cdot u'$. With this notation we have:

Definition 2 (e.s. trace equivalence). Let σ_1 and σ_2 be equivalent environments. Given two processes P and Q, we write $(\ \sigma_1\ ,\ \sigma_2\ ) \vdash P \simeq_{\mathtt{tr}} Q$ if whenever $\sigma_1 \triangleright P \overset{s}{\underset{u}{\Longmapsto}} \sigma_1' \triangleright P'$ then there are s', σ_2' and Q' such that $\sigma_2 \triangleright Q \overset{s'}{\underset{u}{\Longmapsto}} \sigma_2' \triangleright Q'$ and σ_1' is equivalent to σ_2', and symmetrically for $\sigma_2 \triangleright Q$.

This definition highlights a major difference between the π-calculus and the spi-calculus. In the π-calculus "exact" correspondence is required between actions of two equivalent processes P and Q, in the sense that if P is capable of an α-action, then Q must be capable of α too. On the contrary, the presence of cryptography in the spi-calculus allows for a "looser" correspondence. In fact, encrypting two different messages with a secret key makes the two messages indistinguishable for any external observer. Hence, for example, the processes $P \stackrel{\mathrm{def}}{=} (\nu k)\overline{c}\langle\{a\}_k\rangle.\mathbf{0}$ and $Q \stackrel{\mathrm{def}}{=} (\nu k)\overline{c}\langle\{b\}_k\rangle.\mathbf{0}$ are equivalent, even though they do not perform the same (process) actions.

Trace equivalence avoids quantification over contexts and only requires considering transitions of the e.s.-lts. Thus, when compared to the contextual definition of may testing, trace equivalence make reasoning on processes much easier. The following theorem ensures that $\simeq_{\mathtt{tr}}$ is a sound and complete characterization of may-testing equivalence $\simeq$. We denote by ε_V the environment that acts as the identity on the set of names V.

Theorem 1. Let P and Q be spi-processes, and let $V = fn(P,Q)$. It holds that $(\ \varepsilon_V\ ,\ \varepsilon_V\ ) \vdash P \simeq_{\mathtt{tr}} Q$ iff $P \simeq Q$.

A similar result holds for barbed equivalence and an environment-sensitive version of bisimulation.

4.2. Sound reasoning principles

Trace equivalence can be used to justify some rules for syntax-driven reasoning, which are at the core of a sound and complete proof system for the spi-calculus [11]. The rules we are going to list are valid for both bisimulation and trace equivalence. Thus, in what follows, we shall generically write $(\ \sigma_1\ ,\ \sigma_2\ ) \vdash P = Q$ to mean that the configurations $\sigma_1 \triangleright P$ and $\sigma_2 \triangleright Q$ are equivalent, without specifying the actual equivalence.

Structural laws. Table 2 lists a few fundamental equations, mostly inherited from the π-calculus [23], that are valid for any "reasonable" process equivalence. Most of them have to do with "static" structure of processes. Usually, the last three equations are not included in structural equivalence; we have included them here because they are natural in a cryptographic setting. The least equivalence relation over process terms that contains these equations is denoted by $\equiv$ and called *structural equivalence*. One can easily prove the following rule sound:

Table 2
Structural equivalence

$P + \mathbf{0} \equiv P$	$P + Q \equiv Q + P$
$P + (Q + R) \equiv (P + Q) + R$	
$P \mid \mathbf{0} \equiv P$	$P \mid Q \equiv Q \mid P$
$P \mid (Q \mid R) \equiv (P \mid Q) \mid R$	
$P \mid !P \equiv !P$	
$(\nu b)\mathbf{0} \equiv \mathbf{0}$	
$(\nu a)(\nu b)P \equiv (\nu b)(\nu a)P$	
$((\nu a)P) \mid Q \equiv (\nu a)(P \mid Q)$ if $a \notin fn(Q)$	
$[M = M]P \equiv P$	$(\nu n)[n = M]P \equiv \mathbf{0}$ if $M \neq n$
$\mathsf{case}\ \{N\}_k\ \mathsf{of}\ \{y\}_k\ \mathsf{in}\ P \equiv P[^N/_y]$	
$\mathsf{pair}\ \langle M_1, M_2\rangle\ \mathsf{of}\ \langle x,y\rangle\ \mathsf{in}\ P \equiv P[^{M_1}/_x, ^{M_2}/_y]$	

$$\frac{P \equiv Q}{(\ \sigma\ ,\ \sigma\ ) \vdash P = Q}\,.$$

In our example of Section 4.3 we shall make extensive use of two laws derived from structural equivalence. The first one is the so called *extrusion law*:

$$\text{(EXTR)}\qquad \frac{k \notin fn(Q)}{(\ \sigma\ ,\ \sigma\ ) \vdash ((\nu k)P) \mid Q = (\nu k)(P \mid Q)}\,.$$

It states that, if a restricted name k of P does not occur in a process Q running in parallel with P, then the scope of the restriction can be extended so as to include Q.

The second law we shall use is actually a pair of laws (that we shall globally refer to as (MATCH)) can be derived from the structural laws for the matching predicate $[M = N]$. In what follows, we call *context* a process $C[\ \cdot\ ,\dots,\ \cdot\ ]$ with n "holes" that can be filled with n terms, thus yielding a proper process:

(MATCH)

$$(\ \sigma\ ,\ \sigma\ ) \vdash\ C[\ P + [M = M]Q\ ] =\ C[\ P + Q\ ]$$

$$\frac{M \text{ is not a name bound by } (\nu n)\, C[\ \cdot\ ]}{(\ \sigma\ ,\ \sigma\ ) \vdash (\nu n)\, C[\ P + [n = M]Q\ ] = (\nu n)\, C[\ P\ ]}\ .$$

Transitivity. We shall also widely use the obvious transitivity rule:

(TRANS)

$$\frac{(\ \sigma_1\ ,\ \sigma_2\ ) \vdash P = Q \quad \wedge \quad (\ \sigma_2\ ,\ \sigma_3\ ) \vdash Q = R}{(\ \sigma_1\ ,\ \sigma_3\ ) \vdash P = R}\ .$$

Parallel composition. The spi-representation of a security protocol is usually built up by putting in parallel a few simple spi-processes, corresponding to the principals involved in the protocol. A desirable property of each process calculus is that equivalence proofs can be done *compositionally*, i.e. by proving equivalences between subprocesses and then combining together such partial results to get the wanted claim. Unluckily, observational equivalences on the of spi-calculus are not closed under some operators, notably parallel composition. In particular, a naive law like

$$\frac{(\ \sigma_1\ ,\ \sigma_2\ ) \vdash P = Q \quad \wedge \quad (\ \sigma_1\ ,\ \sigma_2\ ) \vdash R = S}{(\ \sigma_1\ ,\ \sigma_2\ ) \vdash P\,|\,R = Q\,|\,S}$$

is not valid. This is due to the interplay between cryptography and private names. As we have already shown at the beginning of Subsection 4.1, a private name k can be extruded and hence become free, without this implying that k is learnt by any observer. As a consequence, we are sometimes confronted with equivalences like: $(\ \sigma_1\ ,\ \sigma_2\ ) \vdash \overline{c}\langle\{a\}_k\rangle.P_1 = \overline{c}\langle\{b\}_k\rangle.P_2$ where both σ_1 and σ_2 know a, b and c, but neither knows k. In general, this kind of equations are not preserved by parallel composition. For instance, when putting $R \overset{\text{def}}{=} \overline{c}\langle k\rangle.\mathbf{0}$ in parallel to both sides of the previous relation, the equivalence breaks down. The reason is that R may provide an observer with the key k to open $\{a\}_k$ and $\{b\}_k$, thus enabling a distinction between these two messages. Similar problems arise from the output prefix (see [11] for a general discussion about problems arising with compositional techniques in the spi-calculus). Fortunately, a more restrictive formulation does hold. Let us denote by $R\sigma$ the result of replacing each name x occurring free in R by $\sigma(x)$. Then we have:

(PAR)

$$\frac{(\ \sigma_1\ ,\ \sigma_2\ ) \vdash P = Q}{(\ \sigma_1\ ,\ \sigma_2\ ) \vdash P\,|\,R\sigma_1 = Q\,|\,R\sigma_2}$$

$$\text{if } fn(R) \subseteq \text{dom}(\sigma_1) = \text{dom}(\sigma_2)\,.$$

The side condition reduces the set of processes that can be composed with P and Q, by requiring that the composed processes are consistent with the knowledge available to σ_1 and σ_2. In spite of this limitation, the rule allows for non trivial forms of compositional reasoning, as shown in [11].

case **elimination.** A common situation for an agent involved in a protocol is waiting for a message and then trying to decrypt it using a key k. This is written as $P \overset{\text{def}}{=} p(x).\mathsf{case}\, x\, \mathsf{of}\, \{y\}_k\, \mathsf{in}\, P'$. Now, suppose that, in some configuration, P comes equipped with an environment $\sigma \overset{\text{def}}{=} \sigma'[\{b\}_k/w]$, such that neither k nor $\{\cdot\}_k$ appears in σ'. Before P evolves, the only message of the form $\{\cdot\}_k$ that σ can produce is $\{b\}_k$. In other words the only message P can receive and then properly decrypt using k is $\{b\}_k$. Thus the behaviour of P in σ is equivalent to $p(x).[x = \{b\}_k]Q[b/y]$. The rule below generalizes this reasoning. We use the notation $\sum_{i=1}^{n} P_i$ to denote the process $P_1 + \ldots + P_n$ (this notation is well-defined since the non-deterministic choice is associative).

(CASE)

$$\begin{aligned}(\ \sigma\ ,\ \sigma\ ) \vdash\ (\nu\widetilde{h},k)\ \Big(&\ C[\ \{M_1\}_k, \ldots, \{M_n\}_k\ ]\ |\\ &D[\ \mathsf{case}\, x\, \mathsf{of}\, \{y\}_k\, \mathsf{in}\ Q\ ]\ \Big) \ = \\ (\nu\widetilde{h},k)\ \Big(&\ C[\ \{M_1\}_k, \ldots, \{M_n\}_k\ ]\ |\\ &D[\ \textstyle\sum_{i=1}^{n}[x = \{M_i\}_k]\ Q[M_i/y]\ ]\Big)\end{aligned}$$

If k does not occur in contexts $C[\ \cdot, \ldots, \cdot\ ]$ and $D[\ \cdot\ ]$ and $\forall\ i = 1, \ldots, n$ C does not bind names in M_i .

4.3. The Kerberos example

Specification. For the sake of readability, we will use in the sequel a few obvious notational shorthands. For example $a(\langle y,z\rangle).P$ stands for $a(x).\mathsf{pair}\ x\ \mathsf{of}\ \langle y,z\rangle\ \mathsf{in}\ P$, $a(\{M\}_k).P$ stands for $a(x).\mathsf{case}\, x\, \mathsf{of}\, \{y\}_k\, \mathsf{in}\, [y = M]P$, and $a(\{M,N\}_k).P$ stands for $a(x).\mathsf{case}\, x\, \mathsf{of}\, \{y\}_k\, \mathsf{in}\ \ \mathsf{pair}\ y\ \mathsf{of}\ \langle z,t\rangle\ \mathsf{in}\ [z = M, t = N]\,P$.
Table 3 gives a high level specification of the protocol using these abbreviations, while Table 4 gives its translation into the syntax of Table 1. All bound names in K are assumed to be distinct from one another and from the free names. Subscripts should help reminding the expected value of each input variable. For instance, the expected value for x_{certB} is B's certificate, i.e. $\{T, k_{AB}, A\}_{k_{BS}}$. Names A and B present in K refer the identity of the principals involved; names *inA* and *reB* are symbolic names that refer the processes associated to A and B respectively (i.e. the principal named A is the initiator of the protocol, while the principal named B is the responder). We decided to keep these names different in order to better distinguish between the principals and the code implementing them.
When starting the protocol execution, all the principals implicitly synchronize on the current time T ($\overline{clock}\langle T\rangle$). This is an approximation of what happens, as the spi-calculus does not provide explicit timing constructs implementing secure clock synchronization (a difficult task which may require complex interactions). Note that *reB* checks the presence of the timestamp T in the first received message and rejects any message not containing T.

Table 3
The Kerberos protocol in spi-calculus

$$
\begin{array}{rcl}
inA & \overset{\mathrm{def}}{=} & \overline{c_{AS}}\langle A,B\rangle.\, c_{AS}(\{T,x_{k_{AB}},B,x_{cert_B}\}_{k_{AS}}).\overline{c_{AB}}\langle x_{cert_B},\{A,n_A\}_{x_{k_{AB}}}\rangle.\, c_{AB}(\{n_A\}_{x_{k_{AB}}}).\,\overline{commit_A}\langle\rangle.\,\mathbf{0}\\
reB & \overset{\mathrm{def}}{=} & c_{AB}(\{T,y_{k_{AB}},A\}_{k_{BS}},\{A,y_{n_A}\}_{y_{k_{AB}}}).\,\overline{c_{AB}}\langle\{y_{n_A}\}_{y_{k_{AB}}}\rangle.\,\overline{commit_B}\langle\rangle.\,\mathbf{0}\\
S & \overset{\mathrm{def}}{=} & c_{AS}(A,B).\overline{c_{AS}}\langle\{T,k_{AB},B,\{T,k_{AB},A\}_{k_{BS}}\}_{k_{AS}}\rangle.\,\mathbf{0}\\
L & \overset{\mathrm{def}}{=} & \overline{lost}\langle\{T_{\mathtt{old}},k_{\mathtt{old}},A\}_{k_{BS}},k_{\mathtt{old}}\rangle.\,\mathbf{0}\\
C & \overset{\mathrm{def}}{=} & \overline{clock}\langle T\rangle.\,\mathbf{0}\\
K & \overset{\mathrm{def}}{=} & (\nu\, k_{AS},k_{BS})\Big(\; L \;\mid\; (\nu\, T)\Big(\; C \;\mid ((\nu\, n_A)inA) \mid reB \mid ((\nu\, k_{AB})S) \;\Big)\;\Big)
\end{array}
$$

Table 4
Full details of *inA*, *reB* and *S* for the Kerberos protocol

$$
\begin{array}{rcl}
inA & \overset{\mathrm{def}}{=} & \overline{c_{AS}}\langle A,B\rangle.\,c_{AS}(x_1).\,\mathsf{case}\,x_1\,\mathsf{of}\,\{x_1'\}_{k_{AS}}\ \mathsf{in}\ \ \mathsf{pair}\ x_1'\ \mathsf{of}\ \langle x_T,x_{k_{AB}},x_B,x_{cert_B}\rangle\ \mathsf{in}\\
 & & [x_T=T,x_B=B]\ \overline{c_{AB}}\langle x_{cert_B},\{A,n_A\}_{x_{k_{AB}}}\rangle.\,c_{AB}(x_2).\,[x_2=\{n_A\}_{x_{k_{AB}}}]\,\overline{commit_A}\langle\rangle.\,\mathbf{0}\\
reB & \overset{\mathrm{def}}{=} & c_{AB}(y).\,\mathsf{pair}\ y\ \mathsf{of}\ \langle y_1,y_2\rangle\ \mathsf{in}\ \ \mathsf{case}\,y_1\,\mathsf{of}\,\{y_1'\}_{k_{BS}}\ \mathsf{in}\ \ \mathsf{pair}\ y_1'\ \mathsf{of}\ \langle y_T,y_{k_{AB}},y_A\rangle\ \mathsf{in}\\
 & & [y_T=T,y_A=A]\ \mathsf{case}\,y_2\,\mathsf{of}\,\{y_2'\}_{y_{k_{AB}}}\ \mathsf{in}\ \ \mathsf{pair}\ y_2'\ \mathsf{of}\ \langle y_A',y_{n_A}'\rangle\ \mathsf{in}\\
 & & [y_A'=A]\,\overline{c_{AB}}\langle\{y_{n_A}'\}_{y_{k_{AB}}}\rangle.\,\overline{commit_B}\langle\rangle.\,\mathbf{0}\\
S & \overset{\mathrm{def}}{=} & c_{AS}(z).\,\mathsf{pair}\ z\ \mathsf{of}\ \langle z_A,z_B\rangle\ \mathsf{in}\ \ [z_A=A,z_B=B]\,\overline{c_{AS}}\langle\{T,k_{AB},B,\{T,k_{AB},A\}_{k_{BS}}\}_{k_{AS}}\rangle.\,\mathbf{0}
\end{array}
$$

Outputs at channels $commit_A$ and $commit_B$ are used to signal that *inA* and *reB* have completed successfully the protocol. For readability, we have omitted the messages carried by these two actions, which are irrelevant here. The $\overline{lost}$-output action accounts for the accidental loss of an old session key $k_{\mathtt{old}}$ and of the corresponding certificate for B, $\{T_{\mathtt{old}},k_{\mathtt{old}},A\}_{k_{BS}}$.

Intuitively, everything works well because the long term keys k_{AS} and k_{BS} remain secret. Of course, if an intruder could forge e.g. k_{BS}, it would be possible for him to create a new certificate (with the current timestamp but with a non-authentic key) and it would be impossible for B to detect the event. Note that the system is not specified so as to guarantee that a commit will eventually be reached: we are only interested in checking that no "wrong" commit will ever happen.

Verification. We will consider authentication of the session key: *"B and A only accept the key k_{AB} generated by S"*. Formally, we want to prove that

$$(\ \varepsilon_I\ ,\ \varepsilon_I\ )\vdash K=K_{aut},$$

where ε_I denotes the environment that acts like the identity on the set of names $I\overset{\mathrm{def}}{=}fn(K,K_{aut})=\{clock,\ lost,\ c_{AS},\ c_{BS},\ c_{AB},\ commit_A,\ commit_B,\ A,\ B,\ T_{\mathtt{old}},\ k_{\mathtt{old}}\}$ and K_{aut}, defined below, formalises the desired protocol's behaviour. inA_{aut} and reB_{aut} can commit only upon receipt of the expected k_{AB} generated by S; in fact, note that K_{aut} is obtained from K's definition by adding the matchings $[x_{k_{AB}}=k_{AB}]$ and $[y_{k_{AB}}=k_{AB}]$ in *inA* and *reB* respectively, upon reception of their certificates.

$$
\begin{array}{rcl}
inA_{aut} & \overset{\mathrm{def}}{=} & \overline{c_{AS}}\langle A,B\rangle.\,c_{AS}(\{T,k_{AB},B,x_{cert_B}\}_{k_{AS}}).\\
 & & \overline{c_{AB}}\langle x_{cert_B},\{A,n_A\}_{k_{AB}}\rangle.\,c_{AB}(\{n_A\}_{k_{AB}}).\\
 & & \overline{commit_A}\langle\rangle.\,\mathbf{0}\\
reB_{aut} & \overset{\mathrm{def}}{=} & c_{AB}(\{T,k_{AB},A\}_{k_{BS}},\{A,y_{n_A}\}_{k_{AB}}).\\
 & & \overline{c_{AB}}\langle\{y_{n_A}\}_{k_{AB}}\rangle.\,\overline{commit_B}\langle\rangle.\,\mathbf{0}\\
K_{aut} & \overset{\mathrm{def}}{=} & (\nu\, k_{AS},k_{BS},k_{AB})\Big(\\
 & & \quad L\ \mid (\nu\, T)\Big(\ C\ \mid ((\nu\, n_A)inA_{aut})\mid\\
 & & \quad reB_{aut}\mid S\ \Big)\ \Big)
\end{array}
$$

We will prove the desired equality by applying the laws of Section 4.2. The proof consists of three steps:

(i) By (EXTR), $(\ \varepsilon_I\ ,\ \varepsilon_I\ )\vdash K=(\nu\, k_{AS},\ k_{BS},\ k_{AB},\ n_A,\ T)(L\,|\,C\,|\,inA\,|\,S\,|\,reB)$. By (CASE) applied to case x_1 of $\ldots$ in *inA*, then by structural equivalence (axiom for pair splitting) and finally by (TRANS), we obtain $(\ \varepsilon_I\ ,\ \varepsilon_I\ )\vdash K=(\nu\, k_{AS},\ k_{BS},\ k_{AB},\ n_A,\ T)(L\,|\,C\,|\,inA'\,|\,S\,|\,reB)$, where

$$
\begin{array}{rl}
inA' \overset{\mathrm{def}}{=} & \overline{c_{AS}}\langle A,B\rangle.\,c_{AS}(x_1).\\
 & [x_1=\{T,k_{AB},B,\{T,k_{AB},A\}_{k_{BS}}\}_{k_{AS}}]\\
 & \overline{c_{AB}}\langle\{T,k_{AB},A\}_{k_{BS}}\rangle.\{A,n_A\}_{k_{AB}}\\
 & c_{AB}(x_2).\,[x_2=\{n_A\}_{k_{AB}}]\ \overline{commit_A}\langle\rangle.\,\mathbf{0}\ .
\end{array}
$$

(By (MATCH), we have deleted the tautological matchings $[T = T, B = B]$). We now apply (CASE) to case y_1 of ... in reB and similarly we obtain $(\varepsilon_I, \varepsilon_I) \vdash K = (\nu k_{AS}, k_{BS}, k_{AB}, n_A, T)(L \mid C \mid inA' \mid S \mid reB_1)$ where

$$\begin{aligned} reB_1 \overset{\text{def}}{=} \\ & c_{AB}(y).\mathsf{pair}\ y\ \mathsf{of}\ \langle y_1, y_2\rangle\ \mathsf{in}\ \big(\\ & \quad [y_1 = \{T, k_{AB}, A\}_{k_{BS}}, T = T, A = A] \\ & \quad \mathsf{case}\, y_2\, \mathsf{of}\ \{y_2'\}_{k_{AB}}\ \mathsf{in}\ \mathsf{pair}\ y_2'\ \mathsf{of}\ \langle y_A', y_{n_A}'\rangle\ \mathsf{in} \\ & \quad [y_A' = A]\ \overline{c_{AB}}\langle \{y_{n_A}'\}_{k_{AB}}\rangle.\overline{commit_B}\langle\rangle.\mathbf{0} \quad + \\ & \quad [y_1 = \{T_{\mathtt{old}}, k_{\mathtt{old}}, A\}_{k_{BS}}, T_{\mathtt{old}} = T, A = A] \\ & \quad \mathsf{case}\, y_2\, \mathsf{of}\ \{y_2'\}_{k_{\mathtt{old}}}\ \mathsf{in}\ \mathsf{pair}\ y_2'\ \mathsf{of}\ \langle y_A', y_{n_A}'\rangle\ \mathsf{in} \\ & \quad [y_A' = A]\ \overline{c_{AB}}\langle \{y_{n_A}'\}_{k_{\mathtt{old}}}\rangle.\overline{commit_B}\langle\rangle.\mathbf{0}\ \big) \end{aligned}$$

By (MATCH), we can delete the tautological matchings $[T = T, A = A]$ from the first summand and delete the second summand (the latter is stuck because of the failure of the matching between T and $T_{\mathtt{old}}$). Hence, by (TRANS), we have

$$(\varepsilon_I, \varepsilon_I) \vdash K \;=\; (\nu k_{AS}, k_{BS}, k_{AB}, n_A, T)(L \mid C \mid inA' \mid reB') \qquad (1)$$

where

$$\begin{aligned} reB' \overset{\text{def}}{=}\ & c_{AB}(y).\mathsf{pair}\ y\ \mathsf{of}\ \langle y_1, y_2\rangle\ \mathsf{in} \\ & [y_1 = \{T, k_{AB}, A\}_{k_{BS}}] \\ & \mathsf{case}\, y_2\, \mathsf{of}\ \{y_2'\}_{k_{AB}}\ \mathsf{in} \\ & \mathsf{pair}\ y_2'\ \mathsf{of}\ \langle y_A', y_{n_A}'\rangle\ \mathsf{in}\ [y_A' = A] \\ & \overline{c_{AB}}\langle \{y_{n_A}'\}_{k_{AB}}\rangle.\overline{commit_B}\langle\rangle.\mathbf{0}. \end{aligned}$$

(ii) Similarly, $(\varepsilon_I, \varepsilon_I) \vdash K_{aut} = (\nu k_{AS}, k_{BS}, k_{AB}, n_A, T)(L \mid C \mid inA' \mid S \mid reB_{aut})$. Then, applying (CASE) to case y_1 of ... in reB_{aut}, we obtain $(\varepsilon_I, \varepsilon_I) \vdash K_{aut} = (\nu k_{AS}, k_{BS}, k_{AB}, n_A, T)(L \mid C \mid inA' \mid S \mid reB'_{aut})$ where

$$\begin{aligned} reB'_{aut} \overset{\text{def}}{=} \\ & c_{AB}(y).\mathsf{pair}\ y\ \mathsf{of}\ \langle y_1, y_2\rangle\ \mathsf{in}\ \big(\\ & \quad [y_1 = \{T, k_{AB}, A\}_{k_{BS}}, T = T, A = A] \\ & \quad [k_{AB} = k_{AB}]\ \mathsf{case}\, y_2\, \mathsf{of}\ \{y_2'\}_{k_{AB}}\ \mathsf{in} \\ & \quad \mathsf{pair}\ y_2'\ \mathsf{of}\ \langle y_A', y_{n_A}'\rangle\ \mathsf{in}\ [y_A' = A] \\ & \quad \overline{c_{AB}}\langle \{y_{n_A}'\}_{k_{AB}}\rangle.\overline{commit_B}\langle\rangle.\mathbf{0} \quad + \\ & \quad [y_1 = \{T_{\mathtt{old}}, k_{\mathtt{old}}, A\}_{k_{BS}}, T_{\mathtt{old}} = T, A = A] \\ & \quad [k_{\mathtt{old}} = k_{AB}]\ \mathsf{case}\, y_2\, \mathsf{of}\ \{y_2'\}_{k_{\mathtt{old}}}\ \mathsf{in} \\ & \quad \mathsf{pair}\ y_2'\ \mathsf{of}\ \langle y_A', y_{n_A}'\rangle\ \mathsf{in}\ [y_A' = A] \\ & \quad \overline{c_{AB}}\langle \{y_{n_A}'\}_{k_{\mathtt{old}}}\rangle.\overline{commit_B}\langle\rangle.\mathbf{0}\ \big) \end{aligned}$$

Again by (MATCH), we can delete the tautological matchings from the first summand and delete the second one, obtaining

$$(\varepsilon_I, \varepsilon_I) \vdash K_{aut} \;=\; (\nu k_{AS}, k_{BS}, k_{AB}, n_A, T)(L \mid C \mid inA' \mid S \mid reB') \qquad (2)$$

(iii) The right hand sides of (1) and (2) are the same. Hence by (TRANS), we obtain the desired $(\varepsilon_I, \varepsilon_I) \vdash K = K_{aut}$.

Finally, notice that without the matching $[y_T = T]$ in reB's definition, the equivalence would be broken. In particular, upon receipt of $\{T_{\mathtt{old}}, k_{\mathtt{old}}, A\}_{k_{BS}}$, reB would perform a final $\overline{commit_B}$, which reB_{aut} cannot do. In essence, removing the check $[y_T = T]$ would recreate the well-known attack against the Needham-Schroeder protocol with symmetric encryption (see e.g. [12]).

5. Trace analysis

We outline here a verification method that departs from the concept of observational equivalence discussed in the previous section. The method is based on analysing the execution traces of a single process representing the protocol. Recall that a *trace* is a sequence of I/O events (actions) executable by a given spi-calculus process. Roughly, a sensible way of expressing authentication of A towards B, in our version of Kerberos, is requiring that, in every trace generated by K, B's final input action is preceded by an A's output of the same message, i.e. B will only accept messages originating from A (similarly for authentication in the other direction).

Trace-based formalizations of authentication and secrecy are generally less demanding than equivalence-based formulations, but more amenable to automatic checking. We will say more on pros and cons of the two approaches in Section 6.

A crucial aspect of the trace analysis method is a notion of *symbolic* execution [7] that avoids having to explicitly consider the infinitely many traces generated by the protocol. This form of state-explosion is related to the interaction of each participant with the external environment. Symbolic execution has been implemented as part of a prototype verification tool named STA (*Symbolic Trace Analyzer*) implemented in ML [8].

In the rest of the section we will first outline the model underlying trace analysis, then touch upon the method of symbolic execution and finally re-consider the Kerberos example in the light of trace analysis.

5.1. Overview of the model

The model underlying the trace analysis method is very close in spirit to Dolev-Yao's one [15]. Informally, agents executing the protocol communicate through a network of public channels that are under the control of an adversary, therefore there are no private, secure channels. Sending a message just means handing the message to the adversary. Conversely, receiving a message just means accepting any message among those the adversary can produce. The adversary records all messages that transit over the network, and can produce a message by either replaying an old one, or by combining old messages (e.g. by pairing, encryption and decryption) and/or by generating fresh quantities.

Formally, a state of the system is a pair $s \triangleright P$, called *configuration*: s is a trace of past I/O events (actions), and represents the current adversary's knowledge; P is a spi-term, describing the intended behavior of honest participants. The set of all configurations is denoted by $\mathscr{C}$. The dynamics of configurations is given by a transition relation $\longrightarrow \subseteq \mathscr{C} \times \mathscr{C}$, that describes elementary steps of computations. In Table 5 we report the rules defining the transition relation, for a subset of the language introduced in Section 2. In particular, since we are looking for an automatic

Table 5
Transition relation on configurations ($\longrightarrow$)

(INP)	$s \triangleright \mathsf{a}(x).P \longrightarrow s \cdot \mathsf{a}\langle M\rangle \triangleright P[^{M}/_{x}]$ if $s \vdash M$ and M is closed
(OUT)	$s \triangleright \overline{\mathsf{a}}\langle M\rangle.P \longrightarrow s \cdot \overline{\mathsf{a}}\langle M\rangle \triangleright P$
(CASE)	$s \triangleright \mathsf{case}\,\{M\}_k\,\mathsf{of}\,\{y\}_k\,\mathsf{in}\,P \longrightarrow s \triangleright P[^{M}/_{y}]$
(SPLIT)	$s \triangleright \mathsf{pair}\ \langle M,N\rangle\ \mathsf{of}\ \langle x,y\rangle\ \mathsf{in}\ P \longrightarrow$ $\longrightarrow s \triangleright P[^{M}/_{x}, {}^{N}/_{y}]$
(MATCH)	$s \triangleright [M = M]P \longrightarrow s \triangleright P$
(RES)	$s \triangleright (\nu a)P \longrightarrow s[^{a'}/_{a}] \triangleright P[^{a'}/_{a}]$ if a' is fresh for s
(PAR)	$\dfrac{s \triangleright P \longrightarrow s' \triangleright P'}{s \triangleright P \mid Q \longrightarrow s' \triangleright P' \mid Q}$
plus symmetric version of (PAR).	

method, we have omitted replication, which would make the problem undecidable (see e.g. [16]). Rules (INP) and (OUT) concern sending and receiving messages, respectively. Since sending a message just means handing the message to the adversary, any output action $\overline{\mathsf{a}}\langle M\rangle$ fired by a process is recorded in the adversary's current knowledge s (rule (OUT)). Conversely, receiving a message just means accepting any message among those the adversary can produce. Therefore, in rule (INP) the variable x can be replaced by any message M non-deterministically chosen among those the adversary can synthesize from its current knowledge s. The synthesis of a message M from a set of known messages S is formalized by a deduction relation $\vdash$. Here is a sample of deduction rules defining $\vdash$ (see [7]):

$$\frac{M \in S}{S \vdash M} \qquad \frac{S \vdash M \qquad S \vdash k}{S \vdash \{M\}_k}$$

$$\frac{S \vdash \{M\}_k \qquad S \vdash k}{S \vdash M}$$

The other operational rules in Table 5 govern how a process decrypts a message ($\mathsf{case}\,M\,\mathsf{of}\,\{y\}_k\,\mathsf{in}\,A$), splits a pair ($\mathsf{pair}\ \langle M,N\rangle\ \mathsf{of}\ \langle x,y\rangle\ \mathsf{in}\ A$), compares two messages for equality ($[M = N]A$), handles a new name ($(\nu a)P$) and interleaves execution of parallel threads ($A \mid B$).

It is worthwhile to point out that there is no need for an explicit description of the adversary's behavior, as the latter is wholly determined by its current knowledge – the s in $s \triangleright P$ – and by the deduction relation $\vdash$. This is somehow in contrast with other proposals [21, 26], where the adversary must be explicitly described, but it is comform to [6, 18, 27].

Given a configuration $s \triangleright P$ and a trace s', we say that $s \triangleright P$ *generates* s' if $s \triangleright P \longrightarrow^* s' \triangleright P'$ for some P' ($\longrightarrow^*$ is the reflexive and transitive closure of $\longrightarrow$, i.e. zero or more steps of $\longrightarrow$). We express properties of the protocol in terms of the traces it generates. In particular, we focus on correspondence assertions of the kind:

> for every generated trace, if action β occurs in the trace, then action α must have occurred at some previous point in the trace,

that is concisely written as $\alpha \hookleftarrow \beta$. More accurately, we allow α and β to contain free variables, that may be instantiated to ground values. Thus $\alpha \hookleftarrow \beta$ actually means that *every instance* of β must be preceded by the corresponding instance of α, for every generated trace. We write $s \triangleright P \models \alpha \hookleftarrow \beta$ if the configuration $s \triangleright P$ satisfies this requirement. This kind of assertions is flexible enough to express interesting secrecy and authentication properties. As an example, the final step of many key-establishment protocols consists in A's sending a message of the form $\{N\}_k$ to B, where N is some authentication information, and k the newly established key. A typical property one wants to verify is that any message encrypted with k that is accepted by B at the final step should actually originate from A (this ensures B he is really talking to A, and that k is authentic). If we call $\mathsf{final}_\mathsf{A}$ and $\mathsf{final}_\mathsf{B}$ the labels attached to A's and B's final action, respectively, then the property might be expressed by $\overline{\mathsf{final}_\mathsf{A}}\langle\{x\}_k\rangle \hookleftarrow \mathsf{final}_\mathsf{B}\langle\{x\}_k\rangle$, for x a variable. The scheme also permits expressing secrecy as a reachability property (in the style of [5, 18]): this is further discussed in Section 6.

5.2. Symbolic execution

When synthesizing new messages, the adversary can apply operations like pairing, encryption and generation of fresh names, an arbitrary number of times. Thus the set of messages the adversary can synthesize at any time is actually infinite in general (i.e. if not empty). Any such message can be non-deterministically chosen by the adversary and sent to a participant willing to receive it; therefore every model based on Dolev and Yao's is in principle infinite. Our model makes no exception: in rule (INP) the set of M s.t. $s \vdash M$ is always infinite, and this makes the model infinitely-branching. This can be regarded as a state explosion problem induced by message exchange.

To overcome this problem, the STA tool implements a verification method based on a notion of symbolic execution. A new transition relation (written $\longrightarrow_{\mathsf{s}}$, below) is introduced in order to condense the infinitely many transitions that arise from an input action (rule (INP) in Table 5) into a single, *symbolic* transition. The received message is now represented simply by a free variable, whose set of possible values is constrained as the execution proceeds. Technically, a constraint takes the form of *most general unifier* (mgu), i.e., the most general substitution that makes two expressions equal. The set of traces generated using the symbolic transition relation constitutes the *symbolic model* of the protocol. Differently from the standard model given by $\longrightarrow$, the symbolic model is finite, because each input action just gives rise to one symbolic transition and agents cannot loop.

For a flavor of how symbolic execution works, let us consider an example focusing on shared-key encryption. Suppose that agent P, after receiving a message, tries decrypting this message using key k; if this succeeds and y is the result, the agent checks whether y equals b and, if so, proceeds like P'. This is written as $P \overset{\text{def}}{=} \mathsf{a}(x).\mathsf{case}\, x\, \mathsf{of}\, \{y\}_k\, \mathsf{in}\, [y=b]P'$, for y fresh. Let us explain how the symbolic execution proceeds, starting from the initial configuration $\varepsilon \triangleright P$. After the first input step, in the second step the decryption $\mathsf{case}\, x\, \mathsf{of}\, \{y\}_k\, \mathsf{in} \cdots$ is resolved by unifying x and $\{y\}_k$, which results in the substitution $[\{y\}_k/x]$. In the third step, the equality test $[y=b]$ is in turn resolved by unifying y and b, that results in $[b/y]$. Formally,

$$\begin{aligned}
\varepsilon \triangleright P \quad &\longrightarrow_{\mathsf{s}} \quad \mathsf{a}\langle x\rangle \triangleright \mathsf{case}\, x\, \mathsf{of}\, \{y\}_k\, \mathsf{in}\, [y=b]P' \\
&\longrightarrow_{\mathsf{s}} \quad \mathsf{a}\langle \{y\}_k\rangle \triangleright [y=b]P'[\{y\}_k/x] \\
&\longrightarrow_{\mathsf{s}} \quad \mathsf{a}\langle \{b\}_k\rangle \triangleright P'[\{y\}_k/x][b/y].
\end{aligned}$$

An important point is that symbolic execution actually ignores the deduction relation $\vdash$ and thus may give rise to "inconsistent" symbolic traces. These inconsistencies can be detected and discovered via a refinement procedure described in [7].

The verification method based on symbolic execution is proven sound and complete w.r.t. the standard model, in the sense that every consistent attack detected in the symbolic model (relation $\longrightarrow_{\mathsf{s}}$) corresponds to some attack in the standard model (relation $\longrightarrow$), and vice-versa. In other words, the symbolic model captures all and only the attacks of the standard model. For instance, the method detects type-dependent attacks, which usually escape finite-state analysis, e.g. [22]. In this kind of attacks, the adversary cheats on the type of some messages, e.g. by inserting a nonce where a key is expected according to the protocol description.

5.3. The Kerberos example

We illustrate the trace analysis method and the use of the automatic tool STA on the simplified Kerberos protocol of Section 3. The tool follows the syntax and semantics of the formal model, with a few minor differences. E.g., action prefixing is written `>>`, parallel composition is written `||`, restriction is written `new-in`, while **0** is written `stop`. Output actions are written as `a!M`, while input actions are written as `a?M`. Note that `M` can be a generic message pattern: this means receiving any adversary-generated message whose form matches `M`. To this purpose, we distinguish explicitly between names and variables (the latters, by convention, start by `x`, `y`, ...). Finally, with `<--` we mean the predicate $\hookleftarrow$ and with `[] @ K` the configuration $\varepsilon \triangleright K$. What follows is the complete STA script defining one session of Kerberos, and the desired authentication properties. Since all channels are public and controlled by the environment, we have made all channel names distinct and used them as references for process actions. Also, we need not make commit actions explicit now, thus we have dropped them.

`Conf` is the initial configuration of the protocol, composed by an empty list of actions and by `K` while `AuthKey`, `AuthAtoB` and `AuthBtoA` represent the properties we want to check of this configuration. `AuthKey` states that any message accepted by A at `a2` should originate from S: this implies the adversary cannot fool A into accepting a key different from `kAB`. Property `AuthAtoB` states that any message accepted by B at `b1` should originate from A at `a3`. `AuthBtoA` can be explained similarly. The three properties together guarantee that A and B always talk to each other, and that they agree on the exchanged data (in particular, on the established key), which are authentic.

If we ask STA to check any of the three properties listed above, we get this answer:

```
 >  val it = "No attack was found, 61
    symbolic configurations reached."
         :  string
```

which means that STA has explored the whole symbolic state-space of the protocol, consisting of 61 configurations, without finding any trace violating the property (this exploration takes STA a fraction of a second). Thus there are no attacks on this configuration of the protocol.

Suppose now we modify B so that it omits the check on the freshness of T, i.e. we re-define

```
val reB=b1?({t,ykAB,A}kBS,{A,ynA}KAB) >>
        b2!{ynA}ykAB >> stop;
```

where we have replaced the timestamp `T` by an arbitrary variable `t` in `b1`. STA finds an attack on the property

```
val inA      = nA new-in ( a1!(A,B) >> a2?{T,xkAB,B,xCertB}KAS >>
                  a3!(xCertB, {A,nA}xkAB) >> a4?{nA}xkAB >> stop );
val S        = kAB new-in ( s1?(A,B) >>
                  s2!{T,kAB,B,{T,kAB,A}kBS}kAS >> stop);
val reB      = b1?({T,ykAB,A}kBS,{A,ynA}ykAB) >>
                  b2!{ynA}ykAB >> stop;
val K        = kAS new-in kBS new-in ( lost!(kOld,{TOld,kOld,A}KBS)>>stop ||
                  T new-in ( clock!T >> stop || inA || reB || S ) );
val Conf     = ( [] @ K );
val AuthKey  = (s2!t <-- a2?t);
val AuthAtoB = (a3!u <-- b1?u);
val AuthBtoA = (b2!w <-- a4?w);
```

`AuthAtoB`. The attack is reported under the form of a trace violating the property:

```
>  val it = "An attack was found:
   lost!(kOld,{TOld,kOld,A}kBS).
   clock!T. a1!(A,B).
   b1?({TOld,kOld,A}kBS1,{A,ynA}kOld)
4  symbolic configurations
   reached." :  string
```

The attack is based on the adversary's replaying the old, compromised key `kOld` and the corresponding certificate `{TOld,kOld,A}kBS` acquired thanks to the `lost` action. Note that the trace contains a free variable `ynA`: it can take on any value which is known to the attacker.

6. A comparison of two methods

An important problem left open by current research is that of establishing a precise relationship between the notions of authentication and secrecy conveyed by the two models outlined in the previous sections.

The equivalence-based formalization is seemingly more demanding than the trace-based one. In fact, the former takes into account the overall behaviour of the protocol – including I/O traffic – while the latter takes into account only correspondence between single actions, or exposure of secret data items. Surprisingly, the two notions are formally incomparable: we show below that neither is stronger than the other. Thus, adopting one notion or the other is not a matter of relative strength. We shall confine our discussion to secrecy, but we feel that similar arguments apply in the case of authentication. First of all, let us state more precisely the notions of secrecy we are interested in.

Definition 3 (two notions of secrecy). Let $P(x)$ be a spi-calculus process. We say that:

- $P(x)$ *keeps* x *E-secret* if for every x': $P(x) \simeq P(x')$;
- $P(x)$ *keeps* x *T-secret* if there is no configuration $s' \triangleright P'$ s.t. $\varepsilon \triangleright P(x) \longrightarrow^* s' \triangleright P'$ and $s' \vdash x$.

Now, consider the process $P(x) \stackrel{\text{def}}{=} (\nu k)(a(y).[y = x]\overline{b}\langle\{x\}_k\rangle.\mathbf{0}$. The process $P(x)$ keeps x T-secret (by inspection), but not E-secret. In fact, consider the observer $O \stackrel{\text{def}}{=} \overline{a}\langle x\rangle.b(z).\omega.\mathbf{0}$: we have $P(x)\,|\,O \stackrel{\omega}{\Longrightarrow}$, but *not* $P(x')\,|\,O \stackrel{\omega}{\Longrightarrow}$, hence $P(x) \not\simeq P(x')$ for $x' \neq x$.

On the other hand, consider $Q(x) \stackrel{\text{def}}{=} a(y).([y = x]\overline{b}\langle x\rangle.\mathbf{0}\,|\,!\overline{b}\langle y\rangle.\mathbf{0})$. Clearly, $Q(x)$ does *not* keep x T-secret. However, $Q(x)$ and $Q(x')$ are trace-equivalent, hence testing equivalent, for any x'; this is a consequence of the fact that $\overline{b}\langle x\rangle.\mathbf{0}\,|\,!\overline{b}\langle x\rangle.\mathbf{0} \equiv !\overline{b}\langle x\rangle.\mathbf{0}$.

The above examples show that E-secrecy does not imply T-secrecy, and, conversely, that T-secrecy does not imply E-secrecy.

7. Concluding remarks and related work

We have outlined some recent approaches to the analysis of security protocols, centered around concepts derived from the field of process calculi, such as observational semantics and symbolic transition systems.

Early work on reasoning methods for the spi-calculus was presented in [4], where *framed bisimulation* was introduced as a proof technique, though incomplete, for reasoning on contextual equivalences. The environment sensitive transition system presented here was introduced in [10], and based on that, the complete characterizations of contextual semantics discussed in Section 4 were obtained. Some of the reasoning principles used in this paper were introduced there. A sound and complete proof system is discussed in [11].

Concerning trace analysis, [7] develops the theory underlying the verification tool STA, while [8] presents verification examples and compares the results to those obtained using finite-state methods. Initial work on symbolic analysis is due to Huima [19]. Symbolic techniques are also exploited in [5, 13, 29], but the algorithms they use are quite different from ours.

Another possible approach consists in deriving properties via type systems: example of these techniques are the type systems in [1, 2] for secrecy and in [17] for authentication. When compared to more traditional methods – like CSP-based model checking [21, 26] – major benefits of the equivalence-based approach seem to be a host of syntax-driven reasoning principles and a fully satisfactory formalization of many important properties, including implicit information flow (that may arise due, e.g., to traffic analysis). On the other hand, the equivalence-based method lacks at present automatic verification techniques. Symbolic trace analysis appears to be closer in spirit to model checking, but does not suffer from the state-explosion problems of model checking, which requires considering approximate models, even when the number of protocol sessions is bounded. Moreover, finite-state model checking has proven very effective in practice to find bugs in security protocols, e.g. [22]. Analysis of real-life case-studies could tell whether the approaches derived from the spi-calculus may represent a valid alternative to the established techniques.

Acknowledgements

We are very grateful to the editor for careful reading and useful suggestions.
This work has been partially supported by EU within the FET – Global Computing initiative, project MIKADO IST-2001-32222 and by MIUR project NAPOLI. The funding bodies are not responsible for any use that might be made of the results presented here.

References

[1] M. Abadi, "Secrecy by typing in security protocols", *J. ACM*, vol. 46, no. 5, pp. 749–786, 1999.

[2] M. Abadi and B. Blanchet, "Analyzing security protocols with secrecy types and logic programs", in *POPL'02*. ACM Press, 2002.

[3] M. Abadi and A. D. Gordon, "A calculus for cryptographic protocols: the spi-calculus", *Inform. Comput.*, vol. 148, no. 1, pp. 1–70, 1999.

[4] M. Abadi and A. D. Gordon, "A bisimulation method for cryptographic protocols", *Nord. J. Comput.*, vol. 5, no. 4, pp. 267–303, 1998.

[5] R. M. Amadio and S. Lugiez, "On the reachability problem in cryptographic protocols", in *Proc. CONCUR'00, LNCS*. Springer, 2000, vol. 1877 (full version: RR 3915, INRIA Sophia Antipolis).

[6] D. Bolignano, "Towards a mechanization of cryptographic protocol verification", in *International Conference on Computer Aided Verification, LNCS*. Springer, 1997.

[7] M. Boreale, "Symbolic trace analysis of cryptographic protocols", in *ICALP'01, LNCS*. Springer, 2001, vol. 2076, pp. 667–681.

[8] M. Boreale and M. G. Buscemi, "Experimenting with STA, a tool for automatic analysis of security protocols", in *ACM Symposium on Applied Computing 2002*. ACM Press, 2002.

[9] M. Boreale and R. De Nicola, "Testing equivalence for mobile processes", *Inform. Comput.*, vol. 120, pp. 279–303, 1995.

[10] M. Boreale, R. De Nicola, and R. Pugliese, "Proof techniques for cryptographic processes", in *LICS'99, Proceedings*. IEEE Computer Society Press, 1999, pp. 157–166 (full version to appear in *SIAM J. Comput.*).

[11] M. Boreale and D. Gorla, "On compositional reasoning in the spi-calculus", in *FoSSaCS'02, Proceedings*, M. Nielsen and H. U. Engberg, Eds., *LNCS*. Springer, 2000, vol. 2303, pp. 67–81.

[12] M. Burrows, M. Abadi, and R. Needham, "A logic of authentication", *ACM Trans. Comput. Syst.*, vol. 8, no. 1, pp. 18–36, 1990.

[13] H. Comon, V. Cortier, and J. Mitchell, "Tree automata with one memory, set constraints and ping-pong protocols", in *ICALP'01, LNCS*. Springer, 2001, vol. 2076, pp. 682–693.

[14] R. De Nicola and M. C. B. Hennessy, "Testing equivalence for processes", *Theor. Comput. Sci.*, no. 34, pp. 83–133, 1984.

[15] D. Dolev and A. Yao, "On the security of public-key protocols", *IEEE Trans. Inform. Theory*, vol. 2, no. 29, pp. 198–208, 1983.

[16] N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov, "Undecidability of bounded security protocols", in *Proc. FLOC Worksh. Form. Meth. Secur. Protoc.*, Trento, Italy, 1999.

[17] A. D. Gordon and A. Jeffrey, "Authenticity by typing for security protocols", in *14th IEEE Comput. Secur. Found. Worksh.*, 2001, pp. 145–159.

[18] J. Goubault-Larrecq, "A method for automatic cryptographic protocol verification", in *Proc. 15th IPDPS Workshops, LNCS*. Springer, 2000, vol. 1800, pp. 977–984.

[19] A. Huima, "Efficient infinite-state analysis of security protocols", in *Proc. FLOC Worksh. Form. Meth. Secur. Protoc.*, Trento, Italy, 1999.

[20] J. Kohl and B. Neuman, "The Kerberos network authentication service (version 5)". Internet Request for Comment RFC-1510, 1993.

[21] G. Lowe, "Breaking and fixing the Needham-Schroeder public-key protocol using FDR", in *TACAS'96*, *Proceedings*, T. Margaria and B. Steffen, Eds., *LNCS*. Springer, 1996, vol. 1055, pp. 147–166.

[22] G. Lowe, "A hierarchy of authentication specifications", in *Proc. 10th IEEE Computer of Security Foundations Workshop*. IEEE Computer Society Press, 1997.

[23] R. Milner, "The polyadic π-calculus: a tutorial", in *Logic and Algebra of Specification*, F. L. Hamer, W. Brauer, and H. Schwichtenberg, Eds. Springer, 1993.

[24] R. Milner, J. Parrow, and D. Walker, "A calculus of mobile processes (Part I and II)", *Inform. Comput.*, vol. 100, pp. 1–77, 1992.

[25] R. Milner and D. Sangiorgi, "Barbed bisimulation", in *ICALP'92*, *Proceedings*, W. Kuich, Ed., *LNCS*. Springer, 1992, vol. 623, pp. 685–695.

[26] J. C. Mitchell, M. Mitchell, and U. Stern, "Automated analysis of cryptographic protocols using Murφ", in *Proceedings of Symposium Security and Privacy*. IEEE Computer Society Press, 1997.

[27] L. C. Paulson, "The inductive approach to verifying cryptographic protocols", *J. Comput. Secur.*, no. 6, pp. 85–128, 1998.

[28] D. Pointcheval, "Asymmetric cryptography and practical security", *J. Telecommun. Inform. Technol.*, no. 4, pp. 41–56, 2002.

[29] M. Rusinowitch and M Turuani, "Protocol insecurity with finite number of sessions in NP-complete", in *14th Computer Security Foundations Workshop*. IEEE Computer Society Press, 2001.

Michele Boreale received a Laurea degree in Scienze dell'Informazione from the University of Pisa in 1991, and a Ph.D. degree in computer science from the University "La Sapienza", Rome, in 1995. He has been research associate at the Dipartimento di Scienze dell'Informazione of University "La Sapienza" from February 1996 to July 1999, when he moved to the Dipartimento di Sistemi e Informatica of the University of Florence. His research interests include formal methods for specifying and verifying concurrent and reactive systems; process calculi and behavioral equivalences, particularly from the angle of elucidating their expressive power and finding tractable proof methods.

e-mail: boreale@dsi.unifi.it
Dipartimento di Sistemi e Informatica
Università di Firenze
Firenze, Italy

Daniele Gorla was born in Rome in 1976; he received his master degree "cum laude" in computer science by the University of Rome "La Sapienza" in 2000. He is currently a Ph.D. student in Florence under the supervision of prof. Rocco De Nicola within the Concurrency and Mobility Group, and he collaborates with the University of Rome.
e-mail: gorla@dsi.uniroma1.it
University of Rome, Italy
e-mail: gorla@dsi.unifi.it
Dipartimento di Sistemi e Informatica
Università di Firenze
Firenze, Italy

Paper

Asymmetric cryptography and practical security

David Pointcheval

Abstract — Since the appearance of public-key cryptography in Diffie-Hellman seminal paper, many schemes have been proposed, but many have been broken. Indeed, for many people, the simple fact that a cryptographic algorithm withstands cryptanalytic attacks for several years is considered as a kind of validation. But some schemes took a long time before being widely studied, and maybe thereafter being broken. A much more convincing line of research has tried to provide "provable" security for cryptographic protocols, in a complexity theory sense: if one can break the cryptographic protocol, one can efficiently solve the underlying problem. Unfortunately, very few practical schemes can be proven in this so-called "standard model" because such a security level rarely meets with efficiency. A convenient but recent way to achieve some kind of validation of efficient schemes has been to identify some concrete cryptographic objects with ideal random ones: hash functions are considered as behaving like random functions, in the so-called "random oracle model", block ciphers are assumed to provide perfectly independent and random permutations for each key in the "ideal cipher model", and groups are used as black-box groups in the "generic model". In this paper, we focus on practical asymmetric protocols together with their "reductionist" security proofs. We cover the two main goals that public-key cryptography is devoted to solve: authentication with digital signatures, and confidentiality with public-key encryption schemes.

Keywords — cryptography, digital signatures, public-key encryption, provable security, random oracle model.

1. Introduction

1.1. Motivation

Since the beginning of public-key cryptography, with the seminal Diffie-Hellman paper [24], many suitable algorithmic problems for cryptography have been proposed (e.g. one-way – possibly trapdoor – functions) and many cryptographic schemes have been designed, together with more or less heuristic proofs of their security relative to the intractability of these problems (namely from the number theory, such as the integer factorization, RSA [69], the discrete logarithm [26] and the Diffie-Hellman [24] problems, or from the complexity theory with some $\mathcal{NP}$-complete problems, such as the knapsack [20] problem or the decoding problem of random linear codes [49]). However, most of those schemes have thereafter been broken.

The simple fact that a cryptographic algorithm withstands cryptanalytic attacks for several years is often considered as a kind of validation procedure, but some schemes take a long time before being broken. The best example is certainly the Chor-Rivest cryptosystem [20, 47], based on the knapsack problem, which took more than 10 years to be totally broken [80], whereas before this last attack it was believed to be very hard, since all the classical techniques against the knapsack problems, such as LLL [46], had failed because of the high density of the involved instances. With this example, but also many others, the lack of attacks at some time should never be considered as a security validation of the proposal.

1.2. Provable security and practical security

A completely different paradigm is provided by the concept of "provable" security. A significant line of research has tried to provide proofs in the framework of complexity theory (a.k.a. "reductionist" security proofs [4]): the proofs provide reductions from a well-studied problem (RSA or the discrete logarithm) to an attack against a cryptographic protocol. At the beginning, people just tried to define the security notions required by actual cryptographic schemes, and then to design protocols which achieve these notions. The techniques were directly derived from the complexity theory, providing polynomial reductions. However, their aim was essentially theoretical, and thus they were trying to minimize the required assumptions on the primitives (one-way functions or permutations, possibly trapdoor, etc.) [33, 35, 52, 67]. Therefore, they just needed to exhibit polynomial reductions from the basic assumption on the primitive into an attack of the security notion, in an asymptotic way.

However, such a result has no practical impact on actual security. Indeed, even with a polynomial reduction, one may be able to break the cryptographic protocol within few hours, whereas the reduction just leads to an algorithm against the underlying problem which requires many years. Therefore, those reductions only prove the security when very huge (and thus maybe unpractical) parameters are used, under the assumption that no polynomial time algorithm exists to solve the underlying problem.

For a few years, more efficient reductions were expected, under the denominations of either "exact security" [10] or "concrete security" [57], which provide more practical security results. The perfect situation is reached when one manages to prove that, from an attack, one can describe an algorithm against the underlying problem, with almost the same success probability within almost the same amount of time. We have then achieved "practical security".

Unfortunately, in many cases, provable security is at the cost of an important loss in terms of efficiency for the cryptographic protocol. Thus some models have been proposed, trying to deal with the security of efficient schemes: some concrete objects are identified with ideal (or black-box) ones.

For example, it is by now usual to identify hash functions with ideal random functions, in the so-called "random oracle model", informally introduced by Fiat and Shamir [27], and formalized by Bellare and Rogaway [8]. Similarly, block ciphers are identified with families of truly random permutations in the "ideal cipher model" [7]. A few years ago, another kind of idealization was introduced in cryptography, the black-box group [53, 77], where the group operation is defined by a black-box: a new element necessarily comes from the addition (or the subtraction) of two already known elements. It is by now called the "generic model". Recent works [17, 73] even use many models together to provide some new validations.

1.3. Outline of the paper

In the next section, we explain and motivate more about exact security proofs, and we introduce the notion of the weaker security analyses, the security arguments (in an ideal model). Then, we review the formalism of the most important asymmetric primitives: signatures and public-key encryption schemes. For both, we provide some examples, with some security analyses in the "reductionist" sense.

2. Security proofs and security arguments

2.1. Basic tools

For asymmetric cryptography, and symmetric cryptography as well, no security can be unconditionally guaranteed, except with the one-time pad [75, 81], an unpractical symmetric encryption method. Therefore, for any cryptographic protocol, security relies on a computational assumption: the existence of one-way functions, or permutations, possibly trapdoor. A one-way function is a function f which anyone can easily compute, but given $y = f(x)$ it is computationally intractable to recover x (or any preimage of y). A one-way permutation is a bijective one-way function. For encryption, one would like the inversion to be possible for the recipient only: a trapdoor one-way permutation is a one-way permutation for which a secret information (the trapdoor) helps to invert the function on any point.

Given such objects, and thus computational assumptions, we would like that security only relies on them. The only way to formally prove such a fact is by showing that an attacker against the cryptographic protocol can be used as a sub-part in an algorithm that can break the basic computational assumption.

2.2. "Reductionist" security proofs

In complexity theory, such an algorithm which uses the attacker as a sub-part in a global algorithm is called a reduction. If this reduction is polynomial, we can say that the attack of the cryptographic protocol is at least as hard as inverting the function: if one has a polynomial algorithm, a.k.a. efficient algorithm, to solve the latter problem, one can polynomially solve the former one, and thus efficiently as well.

Therefore, in order to prove the security of a cryptographic protocol, one first need to make precise the security notion one wants the protocol to achieve: which adversary's goal one wants to be intractable, under which kind of attack. At the beginning of the 1980's, such security notions have been defined for encryption [33] and signature [35, 36], and provably secure schemes have been suggested. However, those proofs had only a theoretical impact, because both the proposed schemes and the reductions were completely unpractical. Indeed, the reductions were efficient (i.e. polynomial), and thus a polynomial attack against a cryptosystem would have led to a polynomial algorithm that broke the computational assumption. But this latter algorithm, even polynomial, may require hundreds of years to solve a small instance. For example, let us consider a cryptographic protocol based on integer factoring. Let us assume that one provides a polynomial reduction from the factorization into an attack. But such a reduction may just lead to a factorization algorithm with a complexity in $2^{100}k^{10}$, where k is the bit-size of the integer to factor. This indeed contradicts the assumption that no-polynomial algorithm exists for factoring. However, on a 1024-bit number, it provides an algorithm that requires 2^{130} basic operations, which is much more than the complexity of the best current algorithm, such as NFS [44], which needs less than 2^{100}. Therefore, such a reduction would just be meaningful for numbers above 2048 bits. Concrete examples are given later.

Moreover, most of the proposed schemes were unpractical as well. Indeed, the protocols were polynomial in time and memory, but not efficient enough for practical implementation.

For a few years, people have tried to provide both practical schemes, with practical reductions and exact complexity, which prove the security for realistic parameters, under a well-defined assumption: exact reduction in the standard model (which means in the complexity-theoretic framework). For example, under the assumption that a 1024-bit integer cannot be factored with less than 2^{70} basic operations, the cryptographic protocol cannot be broken with less than 2^{60} basic operations. We will see such an example later.

Unfortunately, as already remarked, practical and even just efficient reductions, in the standard model, can rarely be conjugated with practical schemes. Therefore, one need to make some hypotheses on the adversary: the attack is generic, independent of the actual implementation of some objects:

– of the hash function, in the "random oracle model";

– of the symmetric block cipher, in the "ideal cipher model";

– of the group, in the "generic model".

The "random oracle model" was the first to be introduced in the cryptographic community [8, 27], and has already been widely accepted. Therefore, in the sequel, we focus on security analyses in this model.

2.3. The random oracle model

As said above, efficiency rarely meets with provable security. More precisely, none of the most efficient schemes in their category have been proven secure in the standard model. However, some of them admit security validations under ideal assumptions: the random oracle model is the most widely accepted one.

Many cryptographic schemes use a hash function H (such as MD5 [68] or the American standards SHA-1 [55], SHA-256, SHA-384 and SHA-512 [56]). This use of hash functions was originally motivated by the wish to sign long messages with a single short signature. In order to achieve *non-repudiation*, a minimal requirement on the hash function is the impossibility for the signer to find two different messages providing the same hash value. This property is called *collision-resistance*.

It was later realized that hash functions were an essential ingredient for the security of, first, signature schemes, and then of most cryptographic schemes. In order to obtain security arguments, while keeping the efficiency of the designs that use hash functions, a few authors suggested using the hypothesis that H behaves like a random function. First, Fiat and Shamir [27] applied it heuristically to provide a signature scheme "as secure as" factorization. Then, Bellare and Rogaway [8, 9] formalized this concept in many fields of cryptography: signature and public-key encryption.

In this model, the so-called "random oracle model", the hash function can be formalized by an oracle which produces a truly random value for each new query. Of course, if the same query is asked twice, identical answers are obtained. This is precisely the context of relativized complexity theory with "oracles," hence the name.

About this model, no one has ever been able to provide a convincing contradiction to its practical validity, but just a theoretical counter-example [18] on a clearly wrong design for practical purpose! Therefore, this model has been strongly accepted by the community, and is considered as a good one, in which proofs of security give a good taste of the actual security level. Even if it does not provide a formal proof of security (as in the standard model, without any ideal assumption), it is argued that proofs in this model ensure security of the overall design of the scheme provided that the hash function has no weakness, hence the name "security arguments".

More formally, this model can also be seen as a restriction on the adversary's capabilities. Indeed, it simply means that the attack is generic without considering any particular instantiation of the hash functions.

On the other hand, assuming the tamper-resistance of some devices, such as smart cards, the random oracle model is equivalent to the standard model, which simply requires the existence of pseudo-random functions [32, 51].

As a consequence, almost all the standards bodies by now require designs provably secure, at least in that model, thanks to the security validation of very efficient protocols.

3. A first formalism

In this section we describe more formally what a signature scheme and an encryption scheme are. Moreover, we make precise the security notions one wants the schemes to achieve. This is the first imperative step towards provable security.

3.1. Digital signature schemes

Digital signature schemes are the electronic version of handwritten signatures for digital documents: a user's signature on a message m is a string which depends on m, on public and secret data specific to the user and – possibly – on randomly chosen data, in such a way that anyone can check the validity of the signature by using public data only. The user's public data are called the *public key*, whereas his secret data are called the *private key*. The intuitive security notion would be the impossibility to forge user's signatures without the knowledge of his private key. In this section, we give a more precise definition of signature schemes and of the possible attacks against them (most of those definitions are based on [36]).

3.1.1. Definitions

A signature scheme is defined by the three following algorithms:

- The *key generation algorithm* K. On input 1^k, which is a formal notation for a machine with running time polynomial in k (1^k is indeed k in basis 1), the algorithm K produces a pair $(\mathsf{k_p}, \mathsf{k_s})$ of matching public and private keys. Algorithm K is probabilistic. The input k is called the security parameter. The sizes of the keys, or of any problem involved in the cryptographic scheme, will depend on it, in order to achieve a security level in 2^k (the expected minimal time complexity of any attack).

- The *signing algorithm* Σ. Given a message m and a pair of matching public and private keys $(\mathsf{k_p}, \mathsf{k_s})$, Σ produces a signature σ. The signing algorithm might be probabilistic.

- The *verification algorithm* V. Given a signature σ, a message m and a public key $\mathsf{k_p}$, V tests whether σ is a valid signature of m with respect to $\mathsf{k_p}$. In general, the verification algorithm need not be probabilistic.

3.1.2. Forgeries and attacks

In this subsection, we formalize some security notions which capture the main practical situations. On the one hand, the **goals** of the adversary may be various:

- Disclosing the private key of the signer. It is the most serious attack. This attack is termed *total break*.
- Constructing an efficient algorithm which is able to sign messages with good probability of success. This is called *universal forgery*.
- Providing a new message-signature pair. This is called *existential forgery*.

In many cases this latter forgery, the *existential forgery*, is not dangerous, because the output message is likely to be meaningless. Nevertheless, a signature scheme which is existentially forgeable does not guarantee by itself the identity of the signer. For example, it cannot be used to certify randomly looking elements, such as keys. Furthermore, it cannot formally guarantee the non-repudiation property, since anyone may be able to produce a message with a valid signature.
On the other hand, various **means** can be made available to the adversary, helping her into her forgery. We focus on two specific kinds of attacks against signature schemes: the *no-message attacks* and the *known-message attacks*. In the first scenario, the attacker only knows the public key of the signer. In the second one, the attacker has access to a list of valid message-signature pairs. According to the way this list was created, we usually distinguish many subclasses, but the strongest is the *adaptive chosen-message attack*, where the attacker can ask the signer to sign any message of her choice. She can therefore adapt her queries according to previous answers.
When one designs a signature scheme, one wants to computationally rule out existential forgeries even under adaptive chosen-message attacks. More formally, one wants that the success probability of any adversary $\mathbf{A}$ with a reasonable time is small, where

$$\mathsf{Succ}_{\mathbf{A}} = \Pr\left[\begin{array}{c}(\mathsf{k_p},\mathsf{k_s}) \leftarrow K(1^k), (m,\sigma) \leftarrow \mathbf{A}^{\Sigma_{\mathsf{k_s}}}(\mathsf{k_p}) : \\ V(\mathsf{k_p},m,\sigma) = 1\end{array}\right].$$

We remark that since the adversary is allowed to play an adaptive chosen-message attack, the signing algorithm is made available, without any restriction, hence the oracle notation $\mathbf{A}^{\Sigma_{\mathsf{k_s}}}$. Of course, in its answer, there is the natural restriction that the returned signature has not been obtained from the signing oracle $\Sigma_{\mathsf{k_s}}$ itself.
This above security level is the strongest one that one can formalize in the communication model we consider. We insist on the fact that in the current communication model, we give the adversary complete access to the cryptographic primitive, but as a black-box. She can ask any query of her choice, and the box always answers correctly, in constant time. Such a model does not consider timing attacks [42], where the adversary tries to extract the secrets from the computational time. Some other attacks analyze the electrical energy required by a computation to get the secrets [43], or to make the primitive fail on some computation [11, 15]. They are not captured either by this model.

3.2. Public-key encryption

The aim of a public-key encryption scheme is to allow anybody who knows the public key of Alice to send her a message that she will be the only one able to recover, granted her private key.

3.2.1. Definitions

A public-key encryption scheme is defined by the three following algorithms:

- The *key generation algorithm* K. On input 1^k where k is the security parameter, the algorithm K produces a pair $(\mathsf{k_p},\mathsf{k_s})$ of matching public and private keys. Algorithm K is probabilistic.
- The *encryption algorithm* E. Given a message m and a public key $\mathsf{k_p}$, E produces a ciphertext c of m. This algorithm may be probabilistic. We write $E(\mathsf{k_p},m;r)$ where r, in the probabilistic case, is the random input to E.
- The *decryption algorithm* D. Given a ciphertext c and the private key $\mathsf{k_s}$, D gives back the plaintext m. This algorithm is necessarily deterministic.

3.2.2. Security notions

As for signature schemes, the **goals** of the adversary may be various. The first common security notion that one would like for an encryption scheme is *one-wayness* (OW): with just public data, an attacker cannot get back the whole plaintext of a given ciphertext. More formally, this means that for any adversary $\mathbf{A}$, her success in inverting E without the private key should be negligible over the probability space $\mathbf{M} \times \Omega$, where $\mathbf{M}$ is the message space and Ω is the space of the random coins r used for the encryption scheme, and the internal random coins of the adversary:

$$\mathsf{Succ}_{\mathbf{A}} = \Pr_{m,r}[(\mathsf{k_p},\mathsf{k_s}) \leftarrow K(1^k) : \mathbf{A}(\mathsf{k_p},E(\mathsf{k_p},m;r)) = m].$$

However, many applications require more from an encryption scheme, namely the *semantic security* (IND) [33], a.k.a. *polynomial security/indistinguishability of encryptions*: if the attacker has some information about the plaintext, for example that it is either "yes" or "no" to a crucial query, any adversary should not learn more with the view of the ciphertext. This security notion requires computational impossibility to distinguish between two messages, chosen by the adversary, one of which has been encrypted, with a probability significantly better than one half: her advantage $\mathsf{Adv}_{\mathbf{A}}$, formally defined as

$$2 \times \Pr_{b,r}\left[\begin{array}{l}(\mathsf{k_p},\mathsf{k_s}) \leftarrow K(1^k), (m_0,m_1,s) \leftarrow \mathbf{A}_1(\mathsf{k_p}),\\ c = E(\mathsf{k_p},m_b;r) : \mathbf{A}_2(m_0,m_1,s,c) = b\end{array}\right] - 1,$$

where the adversary $\mathbf{A}$ is seen as a 2-stage attacker $(\mathbf{A}_1,\mathbf{A}_2)$, should be negligible.

A later notion is *non-malleability* (NM) [25]. To break it, the adversary, given a ciphertext, tries to produce a new ciphertext such that the plaintexts are meaningfully related. This notion is stronger than the above semantic security, but it is equivalent to the latter in the most interesting scenario [6] (the CCA attacks, see below). Therefore, we will just focus on one-wayness and semantic security.

On the other hand, an attacker can play many kinds of attacks, according to the **available information**: since we are considering asymmetric encryption, the adversary can encrypt any plaintext of her choice, granted the public key, hence the *chosen-plaintext attack* (CPA). She may furthermore have access to more information, modeled by partial or full access to some oracles: a plaintext-checking oracle which, on input a pair (m,c), answers whether c encrypts the message m. This attack has been named the *plaintext-checking attack* (PCA) [58]; a validity-checking oracle which, on input a ciphertext c, just answers whether it is a valid ciphertext or not. This weak oracle had been enough to break some famous encryption schemes [13, 40], running the so-called reaction attacks [37]; or the decryption oracle itself, which on any ciphertext, except the challenge ciphertext, answers the corresponding plaintext (*non-adaptive* [52]/*adaptive* [67] *chosen-ciphertext attacks*). This latter scenario which allows adaptively chosen ciphertexts as queries to the decryption oracle is the strongest attack, and is named the *chosen-ciphertext attack* (CCA).

Furthermore, multi-user scenarios can be considered where related messages are encrypted under different keys to be sent to many people (e.g. broadcast of encrypted data). This may provide many useful data for an adversary. For example, RSA is well-known to be weak in such a scenario [38, 76], namely with a small encryption exponent, using the Chinese Remainders Theorem. But recent results prove that semantically secure schemes, in the classical sense as described above, remain secure in multi-user scenarios [3, 5], whatever the kind of attacks.

A general study of these security notions and attacks was conducted in [6], we therefore refer the reader to this paper for more details. See also the summary diagram in Fig. 1. However, we can just review the scenarios we will be interested in in the following:

- One-wayness under chosen-plaintext attacks (OW-CPA) – where the adversary wants to recover the whole plaintext from just the ciphertext and the public key. This is the weakest scenario.
- Semantic security under adaptive chosen-ciphertext attacks (IND-CCA) – where the adversary just wants to distinguish which plaintext, between two messages of her choice, has been encrypted, while she can ask any query she wants to a decryption oracle (except the challenge ciphertext). This is the strongest scenario one can define for encryption (still in our communication model), and thus our goal when we design a cryptosystem.

$$\begin{tikzcd}
\text{NM-CPA} \arrow[d] & & \text{NM-CCA} \arrow[ll] \arrow[d, shift right] \\
\text{IND-CPA} \arrow[d] & & \text{IND-CCA} \arrow[ll] \arrow[d] \arrow[u, shift right] \\
\text{OW-CPA} & \text{OW-PCA} \arrow[l] & \text{OW-CCA} \arrow[l]
\end{tikzcd}$$

OW	–	one-wayness
IND	–	indistinguishability of encryptions (a.k.a., semantic security)
NM	–	non-mallearbility
CPA	–	chosen-plaintext attack
PCA	–	plaintext-checking attack
CCA	–	chosen-ciphertext attack

Fig. 1. Relations between the security notions for asymmetric encryption.

4. The basic assumptions

There are two major families in number theory-based public-key cryptography:

1. The schemes based on integer factoring, and on the RSA problem [69].
2. The schemes based on the discrete logarithm problem, and on the Diffie-Hellman problems [24], in any "suitable" group. The first groups in use were cyclic subgroups of $\mathbf{Z}_p^\star$, the multiplicative group of the modular quotient ring $\mathbf{Z}_p = \mathbf{Z}/p\mathbf{Z}$. But many schemes are now converted on cyclic subgroups of elliptic curves, or of the Jacobian of hyper-elliptic curves, with namely the so-called ECDSA [1], the US Digital Signature Standard [54] on elliptic curves.

4.1. Integer factoring and the RSA problem

The most famous intractable problem is factorization of integers: while it is easy to multiply two prime integers p and q to get the product $n = p \cdot q$, it is not simple to decompose n into its prime factors p and q.

Currently, the most efficient algorithm is based on sieving on number fields. The Number Field Sieve (NFS) method [44] has a complexity in

$$\mathcal{O}(\exp((1.923+o(1))(\ln n)^{1/3}(\ln\ln n)^{2/3})).$$

It has been used to establish the last record, in August 1999, by factoring a 155-digit integer, product of two 78-digit primes [19].
The factored number, called RSA-155, was taken from the "RSA challenge list", which is used as a yardstick for the security of the RSA cryptosystem (see later). The latter is used extensively in hardware and software to protect electronic data traffic such as in the SSL (security sockets layer) Handshake Protocol.
This record is very important since 155 digits correspond to 512 bits. This is the size which is in use in almost all the implementations of the RSA cryptosystem (namely for actual implementations of SSL on the Internet).

```
RSA-155 =
  109417386415705274218097073220403576120\
  037329454492059909138421314763499842889\
  347847179972578912673324976257528997818\
  33797076537244027146743531593354333897
= 102639592829741105772054196573991675900\
  716567808038066803341933521790711307779
* 106603488380168454820927220360012878679\
  207958575989291522270608237193062808643
```

However, this record required thousands of machines, and three months of computation. Furthermore, due to the above complexity of NFS, integer factoring is believed to be a difficult problem, especially for products of two primes of similar sizes larger than 384 bits.
Unfortunately, it just provides a one-way function, without any possibility to invert the process. No information is known to make factoring easier. However, some algebraic structures are based on the factorization of an integer n, where some computations are difficult without the factorization of n, but easy with it: the finite quotient ring $\mathbf{Z}_n$ which is isomorphic to the product ring $\mathbf{Z}_p \times \mathbf{Z}_q$ if $n = p \cdot q$.
For example, the eth power of any element x can be easily computed using the *square-and-multiply* method. It consists in using the binary representation of the exponent $e = e_k e_{k-1} \dots e_0$, computing the successive powers of 2 of x (x^{2^0}, x^{2^1}, ..., x^{2^k}) and eventually to multiply altogether the ones for which $e_i = 1$. However, to compute eth roots, it seems that one requires to know an integer d such that $ed = 1 \bmod \varphi(n)$, where $\varphi(n)$ is the totient Euler function which denotes the cardinality of the multiplicative subgroup $\mathbf{Z}_n^\star$ of $\mathbf{Z}_n$. In the particular case where $n = pq$, $\varphi(n) = (p-1)(q-1)$. And therefore, $ed - 1$ is a multiple of $\varphi(n)$ which is equivalent to the knowledge of the factorization of n [50].
In 1978, Rivest, Shamir and Adleman [69] defined the following problem:

> **The RSA problem.** Let $n = pq$ be the product of two large primes of similar size and e an integer relatively prime to $\varphi(n)$. For a given $y \in \mathbf{Z}_n^\star$, compute the modular eth root x of y (i.e. $x \in \mathbf{Z}_n^\star$ such that $x^e = y \bmod n$.)

The Euler function can be easily computed from the factorization of n, since for any $n = \prod p_i^{v_i}$,

$$\varphi(n) = n \times \prod \left(1 - \frac{1}{p_i}\right).$$

Therefore, with the factorization of n (the trapdoor), the RSA problem can be easily solved. However nobody knows whether the factorization is required, but nobody knows how to do without it either:

> **The RSA assumption.** For any product of two large primes, $n = pq$, large enough, the RSA problem is intractable (presumably as hard as the factorization of n).

4.2. The discrete logarithm and the Diffie-Hellman problems

The setting is quite general: one is given

– a cyclic group $\mathcal{G}$ of prime order q (such as the finite group $(\mathbf{Z}_q, +)$, a subgroup of $(\mathbf{Z}_p^\star, \times)$ for $q|p-1$, or an elliptic curve, etc.);

– a generator $\mathbf{g}$ (i.e. $\mathcal{G} = \langle \mathbf{g} \rangle$).

We note in bold (such as $\mathbf{g}$) any element of the group $\mathcal{G}$, to distinguish it from a scalar $x \in \mathbf{Z}_q$. But such a $\mathbf{g}$ could be an element in $\mathbf{Z}_p^\star$ or a point of an elliptic curve, according to the setting. Above, we talked about a "suitable" group $\mathcal{G}$. In such a group, some of the following problems have to be hard to solve (using the additive notation).

- The **discrete logarithm** problem (**DL**): given $\mathbf{y} \in \mathcal{G}$, compute $x \in \mathbf{Z}_q$ such that $\mathbf{y} = x \cdot \mathbf{g} = \mathbf{g} + \ldots + \mathbf{g}$ (x times), then one writes $x = \log_{\mathbf{g}} \mathbf{y}$.

- The **computational Diffie-Hellman** problem (**CDH**): given two elements in the group $\mathcal{G}$, $\mathbf{a} = a \cdot \mathbf{g}$ and $\mathbf{b} = b \cdot \mathbf{g}$, compute $\mathbf{c} = ab \cdot \mathbf{g}$. Then one writes $\mathbf{c} = \mathbf{DH}(\mathbf{a}, \mathbf{b})$.

- The **decisional Diffie-Hellman** problem (**DDH**): given three elements in the group $\mathcal{G}$, $\mathbf{a} = a \cdot \mathbf{g}$, $\mathbf{b} = b \cdot \mathbf{g}$ and $\mathbf{c} = c \cdot \mathbf{g}$, decide whether $\mathbf{c} = \mathbf{DH}(\mathbf{a}, \mathbf{b})$ (or equivalently, whether $c = ab \bmod q$).

It is clear that they are sorted from the strongest problem to the weakest one. Furthermore, one may remark that they all are "random self-reducible", which means that any instance can be reduced to a uniformly distributed instance: for example, given a specific element $\mathbf{y}$ for which one wants to compute the discrete logarithm x in basis $\mathbf{g}$, one can choose a random $z \in \mathbf{Z}_q$, and compute $\mathbf{z} = z \cdot \mathbf{y}$. The element $\mathbf{z}$ is therefore uniformly distributed in the group, and the discrete logarithm $\alpha = \log_{\mathbf{g}} \mathbf{z}$ leads to $x = \alpha/z \bmod q$. As a consequence, there are only average complexity cases. Thus, the ability to solve a problem for a non-negligible fraction of instances in polynomial time is equivalent to solve any instance in expected polynomial time.

Very recently, Tatsuaki Okamoto and the author [60] defined a new variant of the Diffie-Hellman problem, which we called the *gap Diffie-Hellman problem* (**GDH**), where one wants to solve the **CDH** problem with an access to a **DDH** oracle.

One may easily remark the following properties about above problems:

$$\mathbf{DL} \geq \mathbf{CDH} \geq \{\mathbf{DDH}, \mathbf{GDH}\},$$

where $A \geq B$ means that the problem A is at least as hard as the problem B. However, in practice, no one knows how to solve any of them without breaking the **DL** problem itself. Currently, the most efficient algorithms to solve this latter problem depend on the underlying group. For generic groups (for which no specific algebraic property can be used), algorithms have a complexity in the square root of q, the order of the generator $\mathbf{g}$ [65, 74]. For example, on well-chosen elliptic curves only these algorithms can be used. The last record was established in April 2001 on the curve defined by the equation $y^2 + xy = x^3 + x^2 + 1$ over the finite field with 2^{109} elements.

However, for subgroups of $\mathbf{Z}_p^\star$, some better techniques can be applied. The best algorithm is based on sieving on number fields, as for the factorization. The General Number Field Sieve method [39] has a complexity in

$$\mathcal{O}(\exp((1.923 + o(1))(\ln p)^{1/3}(\ln \ln p)^{2/3})).$$

It was used to establish the last record, in April 2001 as well, by computing discrete logarithms modulo a 120-digit prime. Therefore, 512-bit primes are still safe enough, as far as the generic attacks cannot be used (the generator must be of large order q, at least a 160-bit prime).

For signature applications, one only requires groups where the **DL** problem is hard, whereas encryption needs trapdoor problems and therefore requires groups where some of the **DH**'s problems are also hard to solve.

The **CDH** problem is usually believed to be much stronger than the **DDH** problem, which means that the **GDH** problem is difficult. This was the motivation of our work on new encryption schemes based on the **GDH** problem [58] (see Section 5.3.2).

5. Provably secure designs

5.1. Introduction

Until 1996, no practical **DL**-based cryptographic scheme has ever been formally studied, only heuristically. And surprisingly, at the Eurocrypt'96 conference, two opposite studies were conducted on the El Gamal signature scheme [26], the first **DL**-based signature scheme designed in 1985 and depicted in Fig. 2.

Whereas existential forgeries were known for that scheme, it was believed to prevent universal forgeries. The first analysis, from Daniel Bleichenbacher [12], showed such a universal forgery when the generator g is not properly chosen. The second one, from Jacques Stern and the author [62], proved the security, against existential forgeries under adaptive chosen-message attacks, of a slight variant with a randomly chosen generator g. The slight variant simply replaces the message m by $H(m,r)$ in the computation, while one uses a hash function H that is assumed to behave like a random oracle. It is amazing to remark that the Bleichenbacher's attack also applies on our variant. Therefore, depending on the initialization, our variant could be a very strong signature scheme or become a very weak one!

Initialization	
g a generator of $\mathbf{Z}_p^\star$,	
where p is a large prime	
K**: Key generation**	
private key	$x \in \mathbf{Z}_{p-1}^\star$
public key	$y = g^x \bmod p$
$\rightarrow (y,x)$	
Σ**: Signature of** $m \rightarrow (r,s)$	
K is randomly chosen in $\mathbf{Z}_{p-1}^\star$	
$r = g^K \bmod p$	$s = (m - xr)/K \bmod p-1$
$\rightarrow (r,s)$ is a signature of m	
V**: Verification of** (m,r,s)	
check whether $g^m \overset{?}{=} y^r r^s \bmod p$	

Fig. 2. The El Gamal signature scheme.

As a consequence, a proof has to be performed in details. Furthermore, the conclusions have to be strictly followed by developers, otherwise the concrete implementation of a secure scheme can be very weak.

5.2. Digital signature schemes

5.2.1. History

The first *secure* signature scheme was proposed by Goldwasser *et al.* [35] in 1984. It used the notion of claw-free permutations. A pair of permutations (f,g) is said *claw-free* if it is computationally impossible to find a *claw* (x,y), which satisfies $f(x) = g(y)$. Their proposal provided polynomial algorithms with a polynomial reduction between the research of a claw and an existential forgery under an adaptive chosen-message attack. However, the scheme was totally unpractical. What about practical schemes?

The RSA signature scheme. Two years after the Diffie-Hellman paper [24], Rivest, Shamir and Adleman [69] proposed the first signature scheme based on the "trapdoor one-way permutation paradigm", using the RSA func-

tion: the generation algorithm produces a large composite number $N = pq$, a public key e, and a private key d such that $e \cdot d = 1 \bmod \varphi(N)$. The signature of a message m, encoded as an element in $\mathbf{Z}_N^\star$, is its eth root, $\sigma = m^{1/e} = m^d \bmod N$. The verification algorithm simply checks whether $m = \sigma^e \bmod N$.
However, the RSA scheme is not secure by itself since it is subject to existential forgery: it is easy to create a valid message-signature pair, without any help of the signer, first randomly choosing a certificate σ and getting the signed message m from the public verification relation, $m = \sigma^e \bmod N$.

The Schnorr signature scheme. In 1986 a new paradigm for signature schemes was introduced. It is derived from fair zero-knowledge identification protocols involving a prover and a verifier [34], and uses hash functions in order to create a kind of virtual verifier. The first application was derived from the Fiat-Shamir [27] zero-knowledge identification protocol, based on the hardness of extracting square roots, with a brief outline of its security. Another famous identification scheme [71], together with the signature scheme [72], has been proposed later by Schnorr, based on that paradigm: the generation algorithm produces two large primes p and q, such that $q \geq 2^k$, where k is the security parameter, and $q \,|\, p-1$, as well as an element g in $\mathbf{Z}_p^\star$ of order q. It also creates a pair of keys, the private key $x \in \mathbf{Z}_q^\star$ and the public key $y = g^{-x} \bmod p$. The signature of a message m is a triple (r, e, s), where $r = g^K \bmod p$, with a random $K \in \mathbf{Z}_q$, the "challenge" $e = H(m, r)$ and $s = K + ex \bmod q$. This latter satisfies $r = g^s y^e \bmod p$ with $e = H(m, r)$, which is checked by the verification algorithm. The security results for that paradigm have been considered as folklore for a long time but without any formal validation.

5.2.2. Secure designs

Schnorr's signature and variants. In our papers [62, 63], with Jacques Stern, we formally proved the above paradigm when H is assumed to behave like a random oracle. The proof is based on the by now classical *oracle replay technique*: by a polynomial replay of the attack with different random oracles (the $\mathbf{Q}_i$'s are the queries and the ρ_i's are the answers), we allow the attacker to forge signatures that are suitably related. This generic technique is depicted in Fig. 3, where the signature of a message m is a triple (σ_1, h, σ_2), with $h = H(m, \sigma_1)$ which depends on the message and the first part of the signature, both bound not to change for the computation of σ_2, which really relies on the knowledge of the private key. If the probability of fraud is high enough, then with good probability, the adversary is able to answer to many distinct outputs from the H function, on the input (m, σ_1).

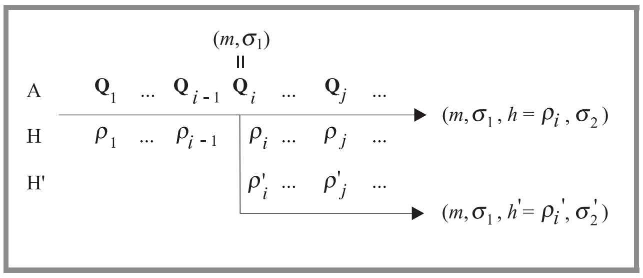

(m,σ_1)
$\parallel$
A
$\mathbf{Q}_1$ ... $\mathbf{Q}_{i-1}$ $\mathbf{Q}_i$ ... $\mathbf{Q}_j$ ...
$(m, \sigma_1, h = \rho_i, \sigma_2)$
H
ρ_1 ... ρ_{i-1} ρ_i ... ρ_j ...
H'
ρ'_i ... ρ'_j ...
$(m, \sigma_1, h' = \rho'_i, \sigma'_2)$

Fig. 3. The oracle replay technique.

Initialization (security parameter k)

$\mathbf{g}$ a generator of any cyclic group $(\mathcal{G}, +)$
of order q, with $2^{k-1} \leq q < 2^k$
H a hash function: $\{0,1\}^\star \to \mathbf{Z}_q$

K**: Key generation**

private key $x \in \mathbf{Z}_q^\star$
public key $\mathbf{y} = -x \cdot \mathbf{g}$
$\to (\mathbf{y}, x)$

Σ**: Signature of** $m \to (\mathbf{r}, h, s)$

K is randomly chosen in $\mathbf{Z}_q^\star$
$\mathbf{r} = K \cdot \mathbf{g}$ $\quad h = H(m, r)$ $\quad s = K + xh \bmod q$
$\to (\mathbf{r}, h, s)$ is a signature of m

V**: Verification of** (m, r, s)

check whether $h \stackrel{?}{=} H(m, \mathbf{r})$
and $\mathbf{r} \stackrel{?}{=} s \cdot \mathbf{g} + h \cdot \mathbf{y}$

Fig. 4. The Schnorr signature scheme.

To be more concrete, let us consider the Schnorr signature scheme, which is presented in Fig. 4, in any "suitable" cyclic group $\mathcal{G}$ of prime order q, where at least the discrete logarithm problem is hard. We expect to obtain two signatures $(\mathbf{r} = \sigma_1, h, s = \sigma_2)$ and $(\mathbf{r}' = \sigma'_1, h', s' = \sigma'_2)$ of an identical message m such that $\sigma_1 = \sigma'_1$, but $h \neq h'$. We then can extract the discrete logarithm of the public key:

$$\left.\begin{array}{rcrcl} \mathbf{r} & = & s \cdot \mathbf{g} & + & h \cdot \mathbf{y} \\ \mathbf{r} & = & s' \cdot \mathbf{g} & + & h' \cdot \mathbf{y} \end{array}\right\} \Rightarrow (s - s') \cdot \mathbf{g} = (h' - h) \cdot \mathbf{y},$$

which leads to

$$\log_{\mathbf{g}} \mathbf{y} = (s - s') \cdot (h' - h)^{-1} \bmod q.$$

Let use denote by ε the success probability of the adversary in performing an existential forgery after q_h queries to the random oracle H. One can prove that for ε large enough, more precisely $\varepsilon \geq 7q_h/2^k$, after less than $16q_h/\varepsilon$ repetitions of this adversary, one can obtain such a pair of signatures with probability greater than $1/9$.
However, this just covers the no-message attacks, which are the weakest attacks! But because we can simulate any zero-knowledge protocol, even without having to restart the simulation since we are in front of an honest verifier (i.e. the

challenge is randomly chosen by the random oracle H) one can easily simulate the signer without the private key:

– one first chooses random $h, s \in \mathbf{Z}_q$;

– one computes $\mathbf{r} = s \cdot \mathbf{g} + h \cdot \mathbf{y}$ and defines $H(m, \mathbf{r})$ to be equal to h, which is a uniformly distributed value;

– one can output $(\mathbf{r}, h, s)$ as a valid signature of the message m.

This furthermore simulates the oracle H, by defining $H(m, \mathbf{r})$ to be equal to h. This simulation is almost perfect since H is supposed to output a random value to any new query, and h is indeed a random value. Nevertheless, if the query $H(m, \mathbf{r})$ has already been asked, $H(m, \mathbf{r})$ is already defined, and thus the definition $H(m, \mathbf{r}) \leftarrow h$ is impossible. But such a situation is very rare, which allows us to claim the following result, which stands for the Schnorr signature scheme but also for any signature derived from a three-round honest verifier zero-knowledge interactive proof of knowledge: let $\mathbf{A}$ be an adversary against the Schnorr signature scheme, with security parameter k, in the random oracle model. We assume that, after q_h queries to the random oracle and q_s queries to the signing oracle, $\mathbf{A}$ produces, with probability $\varepsilon \geq 10(q_s + 1)(q_s + q_h)/2^k$, a valid signature. Then, after less than $23q_h/\varepsilon$ repetitions of this adversary, one can extract the private key x with probability $\varepsilon' \geq 1/9$.

From a more practical point of view, this result states that if an adversary manages to perform an existential forgery under an adaptive chosen-message attack within an expected time T, after q_h queries to the random oracle and q_s queries to the signing oracle, then the discrete logarithm problem can be solved within an expected time less than $207q_hT$.

Brickell, Vaudenay, Yung and the author extended this technique [16, 64] to many variants of El Gamal [26] and DSA [54], such as the Korean Standard KCDSA [41]. However, the original El Gamal and DSA schemes were not covered by this study, and are certainly not provably secure, even if no attack has ever been found against DSA.

RSA-based signatures. Unfortunately, with these signatures, we do not really achieve our goal, because the reduction is costly: if one can break the signature scheme within an expected time T, and q_h queries to the hash function, then one can compute $\log_g y$ within an expected time $207q_hT$, where q_h can be huge, as much as 2^{60} in practice. This security proof is meaningful only for very large groups.

In 1996, Bellare and Rogaway [10] proposed other candidates, based on the RSA assumption. The first scheme is the by now classical hash-and-decrypt paradigm (a.k.a. the Full-Domain Hash paradigm): as for the basic RSA signature, the generation algorithm produces a large composite number $N = pq$, a public key e, and a private key d such that $e \cdot d = 1 \bmod \varphi(N)$. In order to sign a message m, one first hashes it using a full-domain hash function $H: \{0,1\}^\star \rightarrow \mathbf{Z}_N^\star$, and computes the eth root, $\sigma = H(m)^d \bmod N$. The verification algorithm simply checks whether the following equality holds, $H(m) = \sigma^e \bmod N$. For this scheme, they proved, in the random oracle model: if an adversary can produce, with success probability ε, an existential forgery under a chosen-message attack within a time t, after q_h and q_s queries to the hash function and the signing oracle respectively, then the RSA function can be inverted with probability ε' within time t' where

$$\varepsilon' \geq \frac{\varepsilon}{q_s + q_h} \quad \text{and} \quad t' \leq t + (q_s + q_h)T_{exp},$$

with T_{exp} the time for an exponentiation to the power e, modulo N. This reduction has been recently improved [21], thanks to the random self-reducibility of the RSA function, into:

$$\varepsilon' \geq \frac{\varepsilon}{q_s} \times \exp(-1) \quad \text{and} \quad t' \leq t + (q_s + q_h)T_{exp}.$$

This latter reduction works as follows: we assume the existence of an adversary that produces an existential forgery, with probability ε, within time t, after q_h queries to the random oracle H and q_s queries to the signing oracle. We provide this adversary with all the inputs/outputs he needs, while trying to extract the eth root of a given y. For that, we have to simulate the random oracle and the signing oracle.

Simulation of the random oracle H. For any fresh query m to H, one chooses a random $r \in \mathbf{Z}_N^\star$ and flips a biased coin (which returns 0 with probability p, and 1 with probability $1-p$.) If 0 appears, one defines and returns $H(m) = r^e \bmod N$, otherwise one defines and returns $H(m) = yr^e \bmod N$.

Simulation of the signing oracle. For any fresh query m, one first invokes the random oracle on m (if not yet done). If $H(m) = r^e \bmod N$ for some known r, then one returns r as the signature of m, otherwise we stop the simulation and return a failure signal.

At the end of the game, the adversary outputs a valid message/signature $H(m) = \sigma^e \bmod N$. If $H(m)$ has been asked to H during the simulation then, with probability $1-p$, $H(m) = yr^e = \sigma^e \bmod N$ and thus $y = (\sigma/r)^e \bmod N$, which leads to an eth root of y. Otherwise we return a failure signal.

One must first remark that the H simulation is perfect since a new random element in $\mathbf{Z}_N^\star$ is returned for any new query. However, the signing oracle simulation may fail when a signing query is done on a message such that $H(m) = yr^e \bmod N$. Indeed, in this case, the simulation aborts. But such a case happens with probability $1-p$ for any signature. Therefore, the simulation is perfect with probability p^{q_s}, and in such a good case, the forgery leads to the eth root of y with probability $1-p$. Therefore, the success probability of our RSA inversion is $(1-p)p^{q_s}\varepsilon$, which is optimal for $p = 1 - 1/(q_s+1)$. And for this parameter, and a huge value q_s, the success probability is approximately ε/eq_s.

As far as time complexity is concerned, each random oracle simulation (which can be launched by a signing simulation) requires a modular exponentiation to the power e, hence the result.

This is a great improvement since the success probability does not depend anymore on q_h. Furthermore, q_s can be limited by the user, whereas q_h cannot. In practice, one only assumes $q_h \le 2^{60}$, but q_s can be limited below 2^{30}.

However, one would like to get more, suppressing any coefficient. In their paper [10], Bellare and Rogaway proposed such a better candidate, the probabilistic signature scheme (PSS, see Fig. 5): the key generation is still the same, but the signature process involves three hash functions

$$F : \{0,1\}^{k_2} \to \{0,1\}^{k_0}, \quad G : \{0,1\}^{k_2} \to \{0,1\}^{k_1},$$
$$H : \{0,1\}^\star \to \{0,1\}^{k_2},$$

where $k = k_0 + k_1 + k_2 + 1$ is the bit-length of the modulus N. For each message m to be signed, one chooses a random string $r \in \{0,1\}^{k_1}$. One first computes $w = H(m,r)$, $s = G(w) \oplus r$ and $t = F(w)$. Then one concatenates $y = 0\|w\|s\|t$, where $a\|b$ denotes the concatenation of the bit strings a and b. Finally, one computes the eth root, $\sigma = y^d \bmod N$.

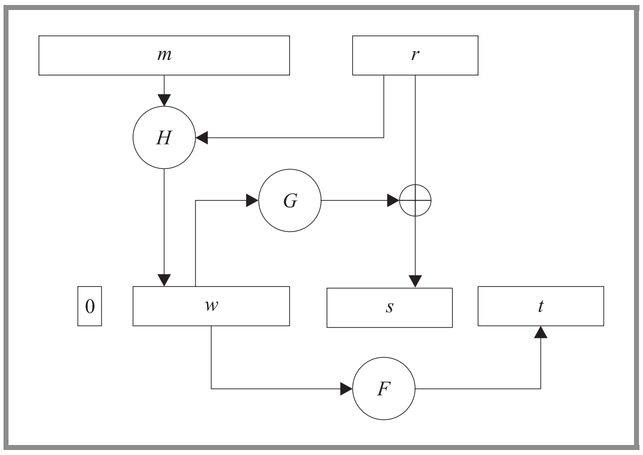


Fig. 5. Probabilistic signature scheme.

The verification algorithm first computes $y = \sigma^e \bmod N$, and parses it as $y = b\|w\|s\|t$. Then, one can get $r = s \oplus G(w)$, and checks whether $b = 0$, $w = H(m,r)$ and $t = F(w)$.

About PSS, Bellare and Rogaway proved, in the random oracle model: if an adversary can produce, with success probability ε, an existential forgery under a chosen-message attack within a time t, after q_h and q_s queries to the hash functions (F, G and H altogether) and the signing oracle respectively, then the RSA function can be inverted with probability ε' within time t' where

$$\begin{array}{rcl} \varepsilon' & \ge & \varepsilon - \dfrac{1}{2^{k_2}} - (q_s + q_H) \cdot \left(\dfrac{q_s}{2^{k_1}} + \dfrac{q_h + q_s + 1}{2^{k_2}} \right) \\ \text{and} \quad t' & \le & t + (q_s + q_H) k_2 T_{exp}, \end{array}$$

with T_{exp} the time for an exponentiation to the power e, modulo N.

The reduction is a bit more intricate than the previous one: once again, we assume the existence of an adversary that produces an existential forgery, with probability ε, within time t, after q_F, q_G, q_H queries to the random oracles F, G, H (we denote $q_h = q_F + q_G + q_H$) and q_s queries to the signing oracle. We provide this adversary with all the inputs/outputs he needs, while trying to extract the eth root of a given y. For that, we have to simulate the random oracles and the signing oracle. For any fresh query (m,r) to the random oracle H, one chooses a random $u \in \mathbf{Z}_N^\star$ and computes $z = yu^e \bmod N$, until the most significant bit of z is 0. Then one parses z into $0\|w\|s\|t$. Then one defines $H(m,r) \leftarrow w$, $G(w) \leftarrow s \oplus r$ and $F(w) \leftarrow t$. One finally returns w.

For any query w to the random oracles F or G, if the answer has not already been defined (during a H simulation) then a random string is returned.

A signing query m is trivially answered: one chooses a random r and runs a specific simulation of $H(m,r)$: one chooses a random $u \in \mathbf{Z}_N^\star$ and computes $z = u^e \bmod N$, until the most significant bit of z is 0. Then one parses z into $0\|w\|s\|t$, and defines $H(m,r) \leftarrow w$, $G(w) \leftarrow s \oplus r$ and $F(w) \leftarrow t$. One finally returns u as a signature of m.

At the end of the game, the adversary outputs a valid message/signature (m,σ), where $\sigma^e = 0\|w\|s\|t \bmod N$, which corresponds to the signature of m with the random $r = G(w) \oplus s$. If $H(m,r)$ has not been asked, $H(m,r) = w$ with probability $1/2^{k_2}$. Therefore

$$\Pr[\mathsf{valid}(m,\sigma) \,|\, \neg\mathsf{AskH}(m,r)] \le 2^{-k_2},$$

where "$\mathsf{valid}(m,\sigma)$" denotes the event that the message/signature (m,σ) is a valid one (and thus accepted by the verification algorithm), and "$\mathsf{AskH}(m,r)$" denotes the event that the query (m,r) has been asked to the random oracle H.

Since this is a forgery, m has never been signed by the signing oracle, and thus $H(m,r)$ has been asked directly by the adversary: $H(m,r) \leftarrow w$, $G(w) \leftarrow s \oplus r$ and $F(w) \leftarrow t$, where $yu^e = 0\|w\|s\|t$, which leads to an eth root of y.

However, the simulations may not be perfect:

- The random oracles simulations may fail if when defining $F(w)$ and $G(w)$, during an H-simulation (a direct one, or the simulation done for the signing simulation), one of them had already been defined before. But this may just occur with probability less than $q_F \cdot 2^{-k_2}$ (because of a previous direct query to F), $q_G \cdot 2^{-k_2}$ (because of a previous direct query to G) or $(q_H + q_s) \cdot 2^{-k_2}$ (because of a previous direct or indirect H-simulation).

- Even after many iterations, the z (computed during the H-simulation, or the signing simulation) may still be greater than $N/2$. We limit this number of iterations to k_2. Then the probability for z to be still greater than $N/2$ is less than $1/2^{k_2}$.

- The signing simulation may fail if the $H(m,r)$ value has already been defined. But this may only occur with probability $(q_H + q_s)2^{-k_1}$.

Therefore, the global success probability in inverting y is greater than

$$\varepsilon - 2^{-k_2} - (q_H + q_s)\left[\frac{q_F + q_G}{2^{k_2}} + \frac{q_H + q_s}{2^{k_2}} + \frac{1}{2^{k_2}} + \frac{q_s}{2^{k_1}}\right],$$

hence the result.
As fas as time complexity is concerned, each H simulation (which can be launched by a signing simulation) requires at most k_2 modular exponentiations to the power e, hence the result. Thanks to this exact and efficient security result, RSA-PSS has become the new PKCS #1 v2.0 standard for signature [70]. Another variant has been proposed with message-recovery. PSS-R allows one to include a large part of the message inside the signature. This makes a signed-message shorter than the size of the signature plus the size of the message, since this latter is inside the former one.

5.3. Public-key encryption

5.3.1. History

The RSA encryption scheme. In the same paper as the RSA signature scheme [69], Rivest, Shamir and Adleman also proposed a public-key encryption scheme, thanks to the "trapdoor one-way permutation" property of the RSA function: the generation algorithm produces a large composite number $N = pq$, a public key e, and a private key d such that $e \cdot d = 1 \bmod \varphi(N)$. The encryption of a message m, encoded as an element in $\mathbf{Z}_N^\star$, is simply $c = m^e \bmod N$. This ciphertext can be easily decrypted thanks to the knowledge of d, $m = c^d \bmod N$. Clearly, this encryption is OW-CPA, relative to the RSA problem. The determinism makes a plaintext-checking oracle useless. Indeed, the encryption of a message m, under a public key $\mathsf{k_p}$ is always the same, and thus it is easy to check whether a ciphertext c really encrypts m, by re-encrypting it. Therefore the RSA-encryption scheme is OW-PCA relative to the RSA problem as well.
Because of this determinism, it cannot be semantically secure: given the encryption c of either m_0 or m_1, the adversary simply computes $c' = m_0^e \bmod N$ and checks whether $c' = c$. Furthermore, with a small exponent e (e.g. $e = 3$), any security vanishes under a multi-user attack: given $c_1 = m^3 \bmod N_1$, $c_2 = m^3 \bmod N_2$ and $c_3 = m^3 \bmod N_3$, one can easily compute $m^3 \bmod N_1 N_2 N_3$ thanks to the Chinese Remainders Theorem, which is exactly m^3 in $\mathbf{Z}$ and therefore leads to an easy recovery of m.

The El Gamal encryption scheme. In 1985, El Gamal [26] also designed a public-key encryption scheme based on the Diffie-Hellman key exchange protocol [24]: given a cyclic group $\mathcal{G}$ of order prime q and a generator $\mathbf{g}$, the generation algorithm produces a random element $x \in \mathbf{Z}_q^\star$ as private key, and a public key $\mathbf{y} = x \cdot \mathbf{g}$. The encryption of a message m, encoded as an element $\mathbf{m}$ in $\mathcal{G}$, is a pair $(\mathbf{c} = a \cdot \mathbf{g}, \mathbf{d} = a \cdot \mathbf{y} + \mathbf{m})$. This ciphertext can be easily decrypted thanks to the knowledge of x, since

$$a \cdot \mathbf{y} = ax \cdot \mathbf{g} = x \cdot \mathbf{c},$$

and thus $\mathbf{m} = \mathbf{d} - x \cdot \mathbf{c}$. This encryption scheme is well-known to be OW-CPA relative to the computational Diffie-Hellman problem. It is also semantically secure (against chosen-plaintext attacks) relative to the decisional Diffie-Hellman problem [79]. For OW-PCA, it relies on the new gap Diffie-Hellman problem [60].

5.3.2. Secure designs

As we have seen above, the expected security level is IND-CCA, whereas the RSA encryption just reaches OW-CPA under the RSA assumption, and the El Gamal encryption achieves IND-CPA under the **DDH** assumption. Can we achieve IND-CCA for practical encryption schemes?

OAEP: the optimal asymmetric encryption padding. In 1994, Bellare and Rogaway proposed a generic conversion [9], in the random oracle model, the optimal asymmetric encryption padding (OAEP, see Fig. 6), which was claimed to apply to any family of trapdoor one-way permutations, such as RSA. The key generation produces a one-way permutation $f : \{0,1\}^k \to \{0,1\}^k$, the public key. The private key is the inverse permutation g, which requires a trapdoor to be computable. The scheme involves two hash functions

$$G : \{0,1\}^{k_0} \to \{0,1\}^{n+k_1}, \quad H : \{0,1\}^{n+k_1} \to \{0,1\}^{k_0},$$

where $k = k_0 + k_1 + n + 1$. For any message $m \in \{0,1\}^n$ to be encrypted. Instead of computing $f(m)$, as done with the above plain-RSA encryption, one first modifies m, choosing a random string $r \in \{0,1\}^{k_0}$. Then one computes $s = (m \| 0^{k_1}) \oplus G(r)$ and $t = r \oplus H(s)$. Finally, one computes $c = f(s \| t)$.

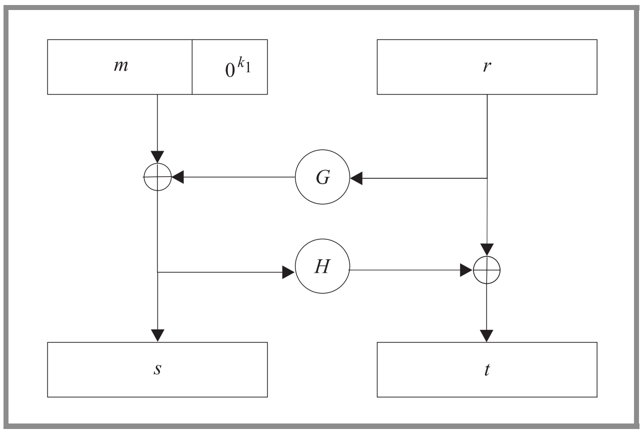


Fig. 6. Optimal asymmetric encryption padding.

The decryption algorithm first computes $P = g(c)$, granted the private key g, and parses it as $P = s||t$. Then, one can get $r = t \oplus H(s)$, and $M = s \oplus G(r)$, which is finally parsed into $M = m||0^{k_1}$, if the k_1 least significant bits are all 0.

For a long time, the OAEP conversion has been widely believed to provide an IND-CCA encryption scheme from any trapdoor one-way permutation. However, the sole proven result (weak plaintext-awareness [9]) was the semantic security against non-adaptive chosen-ciphertext attacks (a.k.a. lunchtime attacks [52]). And recently, Shoup [78] showed that it was very unlikely that a stronger security result could be proven. However, because of the wide belief of a strong security level, RSA-OAEP became the new PKCS #1 v2.0 for encryption [70], and thus a *de facto* standard, after an effective attack against the PKCS #1 v1.5 [13].

Fortunately, Fujisaki, Okamoto, Stern and the author [31] provided a complete security proof: first we proved that combined with a trapdoor partial-domain one-way permutation, the OAEP construction leads to an IND-CCA cryptosystem. A partial-domain one-way permutation f is a one-way permutation such that given $y = f(s||t)$ it is intractable to recover the full $s||t$, but even s only. Furthermore, we provided a complete reduction between the full-domain inversion of RSA and the partial-domain inversion. Therefore, RSA-OAEP really achieves IND-CCA security under the RSA assumption.

The proof is a bit intricate, so we refer the reader to [31] for more information. Anyway, we can claim

> if there exists a CCA-adversary against the "semantic security" of RSA-OAEP (with a k-bit long modulus, with $k > 2k_0$), with running time bounded by t and advantage ε, making q_D, q_G and q_H queries to the decryption oracle, and the hash functions G and H respectively, then the RSA problem could be solved with probability ε', within time bound t', where

$$\begin{aligned}\varepsilon' &\geq \frac{\varepsilon^2}{4} - \varepsilon \cdot \left(\frac{2q_Dq_G + q_D + q_G}{2^{k_0}} + \frac{2q_D}{2^{k_1}} + \frac{32}{2^{k-2k_0}}\right)\\ t' &\leq 2t + q_H \cdot (q_H + 2q_G) \times \mathcal{O}(k^3).\end{aligned}$$

Unfortunately, the reduction is very expensive, and is thus meaningful only for huge moduli, more than 4096-bit long. Indeed, the RSA inverter we can build, thanks to this reduction, has a complexity at least greater than $q_H \cdot (q_H + 2q_G) \times \mathcal{O}(k^3)$. As already remarked, the adversary can ask up to 2^{60} queries to the hash functions, and thus this overhead in the inversion is at least 2^{151}. However, current factoring algorithms can factor up to 4096 bit-long integers within this number of basic operations (see [45] for complexity estimates of the most efficient factoring algorithms).

Anyway, the formal proof shows that the global design of OAEP is sound, and that it is still probably safe to use it in practice (e.g. in PKCS #1 v2.0, while being very careful during the implementation [48]).

More general conversions. Unfortunately, there is no hope to use OAEP with any **DL**-based primitive, because of the "permutation" requirement. The OAEP construction indeed requires the primitive to be a permutation (trapdoor partial-domain one-way), which is the case of the RSA function. However, the only trapdoor problem known in the **DL**-setting is the Diffie-Hellman problem, and it does not provide any bijection. Thus, first Fujisaki and Okamoto [29] proposed a generic conversion from any IND-CPA scheme into an IND-CCA one, in the random oracle model. While applying this conversion to the above El Gamal encryption (see 5.3.1.), one obtains an IND-CCA encryption scheme relative to the **DDH** problem. Later, independently, Fujisaki and Okamoto [30] and the author [61] proposed better generic conversions since they apply to any OW-CPA scheme to make it into an IND-CCA one, still in the random oracle model.

This high security level is just at the cost of two more hashings for the new encryption algorithm, as well as two more hashings but one re-encryption for the new decryption process.

REACT: a rapid enhanced-security asymmetric cryptosystem transform. The re-encryption cost is the main drawback of these conversions for practical purposes. Therefore, Okamoto and the author tried and succeeded in providing a conversion that is both secure and efficient [58]: REACT, for "rapid enhanced-security asymmetric cryptosystem transform".

This latter conversion is indeed very efficient in many senses

- the computational overhead is just the cost of two hashings for *both* encryption and decryption,
- if one can break IND-CCA of the resulting scheme with an expected time T, one can break OW-PCA of the basic scheme within almost the same amount of time, with a low overhead (not as with OAEP). It thus provides a *practical* security result.

Let us describe this generic conversion REACT [58] on any encryption scheme $\mathbf{S} = (K, E, D)$

$$E : \mathbf{PK} \times \mathbf{M} \times \mathbf{R} \to \mathbf{C} \qquad D : \mathbf{SK} \times \mathbf{C} \to \mathbf{M},$$

where **PK** and **SK** are the sets of the public and private keys, **M** is the messages space, **C** is the ciphertexts space and **R** is the random coins space. One should remark that **R** may be small and even empty, with a deterministic encryption scheme, such as RSA. But in many other cases, such as the El Gamal encryption, it is as large as **M**. We also need two hash functions G and H,

$$G : \mathbf{M} \to \{0,1\}^{\ell}, H : \mathbf{M} \times \{0,1\}^{\ell} \times \mathbf{C} \times \{0,1\}^{\ell} \to \{0,1\}^{\kappa},$$

where κ is the security parameter, while ℓ denotes the size of the messages to encrypt. The REACT conversion is depicted in Fig. 7.

K': **Key generation**

$(\mathsf{k_p}, \mathsf{k_s}) \leftarrow K(1^k)$
$\rightarrow (\mathsf{k_p}, \mathsf{k_s})$

E': **Encryption of** $m \in \mathbf{M}' = \{0,1\}^\ell \rightarrow (a,b,c)$

$R \in \mathbf{M}$ and $r \in \mathbf{R}$ are randomly chosen
$a = E(\mathsf{k_p}, R; r) \qquad b = m \oplus G(R) \qquad c = H(R,m,a,b)$
$\rightarrow (a,b,c)$ is the ciphertext

D': **Decryption of** (a,b,c)

Given $a \in \mathbf{C}$, $b \in \{0,1\}^\ell$ and $c \in \{0,1\}^\kappa$
$R = D(\mathsf{k_s}, a) \qquad m = b \oplus G(R)$
if $c = H(R,m,a,b)$ and $R \in \mathbf{M} \rightarrow m$ is the plaintext
(otherwise, "Reject: invalid ciphertext")

Fig. 7. Rapid enhanced-security asymmetric cryptosystem transform $\mathbf{S}'$.

In this new scheme $\mathbf{S}'$, one can claim that if an attacker, against the semantic security in a chosen-ciphertext scenario, can gain an advantage ε after q_D, q_G and q_H queries to the decryption oracle and to the random oracles G and H respectively, within a time t, then one can design an algorithm that outputs, for any given C, the plaintext of C, after less than $q_G + q_H$ queries to the plaintext-checking oracle with probability greater than $\varepsilon/2 - q_D/2^\kappa$, within a time $t + (q_G + q_H)T_{\mathrm{PCA}}$, where T_{PCA} denotes the times required by the PCA oracle to answer any query.
This security result, in the random oracle model, comes from two distinct remarks:

- The adversary has necessarily asked either $G(R)$ or $H(R, m_i, a, b)$ to get any information about the encrypted message m (either m_0 or m_1). Which means that for a given $C = E(\mathsf{k_p}, R; r)$, R is in the list of queries asked to G or to H. Simply asking for the $q_G + q_H$ candidates to the plaintext-checking oracle, one can output the right one. Then, with probability $\varepsilon/2$, one inverts E, after $(q_G + q_H)$ queries to the plaintext-checking oracle.

- However, in the chosen-ciphertext scenario, the adversary may ask queries to the decryption oracle. We have to simulate this. For each query (a,b,c) asked by the adversary to the decryption oracle, one looks at all the pairs (R,m) such that (R,m,a,b) has been asked to the random oracle H. For any such R, one asks the plaintext-checking oracle whether a is a ciphertext of R (remark that it does not make more queries to the plaintext-checking oracle, since it has already been taken into account above). Then it computes $K = G(R)$, maybe using a simulation of G if the query R has never been asked. If $b = K \oplus m$ then one outputs m as the plaintext of the triple (a,b,c). Therefore, any correctly computed ciphertext is decrypted by the simulator. But if the adversary has not asked $H(R,m,a,b)$ the probability that the ciphertext is valid, and thus the decryption not correctly simulated, is less than $1/2^\kappa$.

Hybrid conversion. In this REACT conversion, one can improve efficiency, replacing the one-time pad [81] by any symmetric encryption scheme: indeed, we have computed some $b = m \oplus K$, where $K = G(R)$ can be seen as a session key used in an one-time pad encryption scheme. But one could use any symmetric encryption scheme $(\mathbf{E}, \mathbf{D})$ that is just semantically secure (under no plaintext nor ciphertext attacks). Indeed, the one-time pad achieves perfect semantic security, against this kind of very weak attacks. But one can tolerate some imperfection. Anyway, most of the candidates to the AES process (the call for symmetric encryption schemes, from the NIST, to become the new international standard), and the AES itself (the winner), resisted to more powerful attacks, and thus can be considered strongly secure in our scenario. Therefore, plaintexts of any size could be encrypted using this conversion (see Fig. 8), with a very high speed rate.

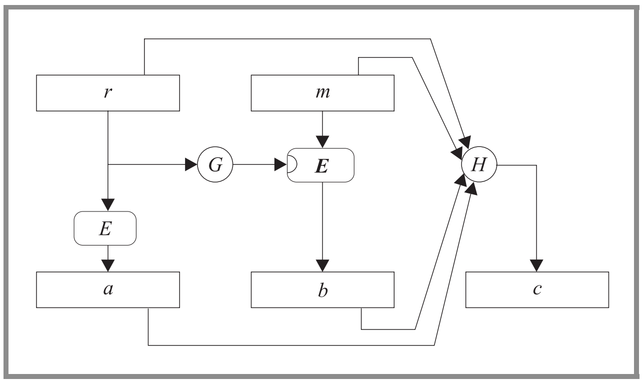


Fig. 8. Hybrid rapid enhanced-security asymmetric cryptosystem transform.

RSA-OAEP alternatives. As we have said, RSA-OAEP has become a *de facto* standard, even if its security has recently been subject to controversy. However, the practical security, for usual sizes (between 512 and 1024 bits), is not really proven because of the huge overhead in the reduction. Some alternatives have been proposed, such as OAEP+ [78] and SAEP(+) [14], but still with expensive reductions, in the general RSA context (some efficient reductions have been proposed for OAEP or SAEP, but only with RSA exponent 3, or the Rabin primitive [66]). Therefore RSA-REACT [59] looks like the best alternative to RSA-OAEP, thanks to the efficient reduction, and the provable security relative the RSA assumption (in the random oracle model).

6. Conclusion

Recently, Cramer and Shoup proposed the first schemes, for both encryption [22] and signature [23], with formal security proofs in the standard model (without any ideal assumption). The encryption scheme achieves IND-CCA under the sole **DDH** assumption, which says that the **DDH** problem is intractable. The signature scheme prevents existential forgeries, even against adaptive chosen-message attacks, under the Strong RSA assumption [2, 28], which claims the intractability of the Flexible RSA problem:

> Given an RSA modulus N and any $y \in \mathbf{Z}_N^\star$, produce x and a prime integer e such that $y = x^e \bmod N$.

Both schemes are very nice because they are the first efficient schemes with formal security proofs in the standard model. However, we have not presented them, nor the reductions either. Actually, they are very intricate and pretty expensive. Furthermore, even if no ideal assumptions are required, the complexity of the reductions make them meaningless for practical parameters. Moreover, even if the schemes are much more efficient than previous proposals in the standard model, they are still much more than twice as expensive as the schemes presented along this paper, in the random oracle model. This is enough to rule them out from practical use. Indeed, everybody wants security, but only if it is quite transparent (and particularily from the financial point of view). Therefore, provable security must not decrease efficiency. It is the reason why strong security arguments, under a realistic restriction on the adversary's capabilities, for efficient schemes have a more practical impact than security proofs in the standard model for less efficient schemes. Of course, quite efficient schemes with formal security proofs are still the target, and thus an exciting challenge.

References

[1] American National Standards Institute. Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm. ANSI X9.62-1998. Jan. 1999.

[2] N. Barić and B. Pfitzmann, "Collision-free accumulators and fail-stop signature schemes without trees", in *Advances in Cryptology – Proceedings of EUROCRYPT '97*, W. Fumy, Ed., *Lecture Notes in Computer Science*. Berlin: Springer, 1997, vol. 1233, pp. 480–484.

[3] O. Baudron, D. Pointcheval, and J. Stern, "Extended notions of security for multicast public key cryptosystems", in *Proceedings of the 27th International Colloquium on Automata, Languages and Programming (ICALP '2000)*, J. D. P. R. U. Montanari and E. Welzl, Eds., *Lecture Notes in Computer Science*. Berlin: Springer, 2000, vol. 1853, pp. 499–511.

[4] M. Bellare, "Practice-oriented provable security", in *Proceedings of First International Workshop on Information Security (ISW '97)*, E. Okamoto, G. Davida, and M. Mambo, Eds., *Lecture Notes in Computer Science*. Berlin: Springer, 1997, vol. 1396.

[5] M. Bellare, A. Boldyreva, and S. Micali, "Public-key encryption in a multi-user setting: security proofs and improvements", in *Advances in Cryptology – Proceedings of EUROCRYPT '2000*, B. Preneel, Ed., *Lecture Notes in Computer Science*. Berlin: Springer, 2000, vol. 1807, pp. 259–274.

[6] M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway, "Relations among notions of security for public-key encryption schemes", in *Advances in Cryptology – Proceedings of CRYPTO '98*, H. Krawczyk, Ed., *Lecture Notes in Computer Science*. Berlin: Springer, 1998, vol. 1462, pp. 26–45.

[7] M. Bellare, D. Pointcheval, and P. Rogaway, "Authenticated key exchange secure against dictionary attacks", in *Advances in Cryptology – Proceedings of EUROCRYPT '2000*, B. Preneel, Ed., *Lecture Notes in Computer Science*. Berlin: Springer, 2000, vol. 1807, pp. 139–155.

[8] M. Bellare and P. Rogaway, "Random oracles are practical: a paradigm for designing efficient protocols", in *Proceedings of the 1st ACM Conference on Computer and Communications Security*. New York: ACM Press, 1993, pp. 62–73.

[9] M. Bellare and P. Rogaway, "Optimal asymmetric encryption – how to encrypt with RSA", in *Advances in Cryptology – Proceedings of EUROCRYPT '94*, A. D. Santis, Ed., *Lecture Notes in Computer Science*. Berlin: Springer, 1995, vol. 950, pp. 92–111.

[10] M. Bellare and P. Rogaway, "The exact security of digital signatures – how to sign with RSA and Rabin", in *Advances in Cryptology – Proceedings of EUROCRYPT '96*, U. Maurer, Ed., *Lecture Notes in Computer Science*. Berlin: Springer, 1996, vol. 1070, pp. 399–416.

[11] E. Biham and A. Shamir, "Differential fault analysis of secret key cryptosystems", in *Advances in Cryptology – Proceedings of CRYPTO '97*, B. Kaliski, Ed., *Lecture Notes in Computer Science*. Berlin: Springer, 1997, vol. 1294, pp. 513–525.

[12] D. Bleichenbacher, "Generating El Gamal signatures without knowing the secret key", in *Advances in Cryptology – Proceedings of EUROCRYPT '96*, U. Maurer, Ed., *Lecture Notes in Computer Science*. Berlin: Springer, 1996, vol. 1070, pp. 10–18.

[13] D. Bleichenbacher, "A chosen ciphertext attack against protocols based on the RSA encryption standard PKCS #1", in *Advances in Cryptology – Proceedings of CRYPTO '98*, H. Krawczyk, Ed., *Lecture Notes in Computer Science*. Berlin: Springer, 1998, vol. 1462, pp. 1–12.

[14] D. Boneh, "Simplified OAEP for the RSA and Rabin functions", in *Advances in Cryptology – Proceedings of CRYPTO '2001*, J. Kilian, Ed., *Lecture Notes in Computer Science*. Berlin: Springer, 2001, vol. 2139, pp. 275–291.

[15] D. Boneh, R. DeMillo, and R. Lipton, "On the importance of checking cryptographic protocols for faults", in *Advances in Cryptology – Proceedings of EUROCRYPT '97*, W. Fumy, Ed., *Lecture Notes in Computer Science*. Berlin: Springer, 1997, vol. 1233, pp. 37–51.

[16] E. Brickell, D. Pointcheval, S. Vaudenay, and M. Yung, "Design validations for discrete logarithm based signature schemes", in *Workshop on Practice and Theory in Public-Key Cryptography (PKC '2000)*, H. Imai and Y. Zheng, Eds., *Lecture Notes in Computer Science*. Berlin: Springer, 2000, vol. 1751, pp. 276–292.

[17] D. R. L. Brown and D. B. Johnson, "Formal security proofs for a signature scheme with partial message recovery", in *The Cryptographers' Track at RSA Conference '2001 (RSA '2001)*, D. Naccache, Ed., *Lectures Notes in Computer Science*. Berlin: Springer, 2001, vol. 2020, pp. 126–142.

[18] R. Canetti, O. Goldreich, and S. Halevi, "The random oracles methodology, revisited", in *Proceedings of the 30th ACM Symposium on the Theory of Computing (STOC '98)*. New York: ACM Press, 1998, pp. 209–218.

[19] S. Cavallar, B. Dodson, A. K. Lenstra, W. Lioen, P. L. Montgomery, B. Murphy, H. te Riele, K. Aardal, J. Gilchrist, G. Guillerm, P. Leyland, J. Marchand, F. Morain, A. Muffett, C. Putnam, C. Putnam, and P. Zimmermann, "Factorization of a 512-bit RSA modulus", in *Advances in Cryptology – Proceedings of EUROCRYPT '2000*, B. Preneel, Ed., *Lecture Notes in Computer Science*. Berlin: Springer, 2000, vol. 1807, pp. 1–18.

[20] B. Chor and R. L. Rivest, "A knapsack type public key cryptosystem based on arithmetic in finite fields", in *Advances in Cryptology – Proceedings of CRYPTO '84*, B. Blakley and D. Chaum, Eds., *Lecture Notes in Computer Science*. Berlin: Springer, 1985, vol. 196, pp. 54–65.

[21] J.-S. Coron, "On the exact security of full-domain-hash", in *Advances in Cryptology – Proceedings of CRYPTO '2000*, M. Bellare, Ed., *Lecture Notes in Computer Science*. Berlin: Springer, 2000, vol. 1880, pp. 229–235.

[22] R. Cramer and V. Shoup, "A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack", in *Advances in Cryptology – Proceedings of CRYPTO '98*, H. Krawczyk, Ed., *Lecture Notes in Computer Science*. Berlin: Springer, 1998, vol. 1462, pp. 13–25.

[23] R. Cramer and V. Shoup, "Signature scheme based on the Strong RSA assumption", in *Proceedings of the 6th ACM Conference on Computer and Communications Security*. New York: ACM Press, 1999, pp. 46–51.

[24] W. Diffie and M. E. Hellman, "New directions in cryptography", *IEEE Trans. Inform. Theory*, vol. IT–22, no. 6, pp. 644–654, 1976.

[25] D. Dolev, C. Dwork, and M. Naor, "Non-malleable cryptography", *SIAM J. Comput.*, vol. 30, no. 2, pp. 391–437, 2000.

[26] T. El Gamal, "A public key cryptosystem and a signature scheme based on discrete logarithms", *IEEE Trans. Inform. Theory*, vol. IT–31, no. 4, pp. 469–472, 1985.

[27] A. Fiat and A. Shamir, "How to prove yourself: practical solutions of identification and signature problems", in *Advances in Cryptology – Proceedings of CRYPTO '86*, A. M. Odlyzko, Ed., *Lecture Notes in Computer Science*. Berlin: Springer, 1987, vol. 263, pp. 186–194.

[28] E. Fujisaki and T. Okamoto, "Statistical zero knowledge protocols to prove modular polynomial relations", in *Advances in Cryptology – Proceedings of CRYPTO '97*, B. Kaliski, Ed., *Lecture Notes in Computer Science*. Berlin: Springer, 1997, vol. 1294, pp. 16–30.

[29] E. Fujisaki and T. Okamoto, "How to enhance the security of public-key encryption at minimum cost", in *Workshop on Practice and Theory in Public-Key Cryptography (PKC '99)*, H. Imai and Y. Zheng, Eds., *Lecture Notes in Computer Science*. Berlin: Springer, 1999, vol. 1560, pp. 53–68.

[30] E. Fujisaki and T. Okamoto, "Secure integration of asymmetric and symmetric encryption schemes", in *Advances in Cryptology – Proceedings of CRYPTO '99*, M. Wiener, Ed., *Lecture Notes in Computer Science*. Berlin: Springer, 1999, vol. 1666, pp. 537–554.

[31] E. Fujisaki, T. Okamoto, D. Pointcheval, and J. Stern, "RSA-OAEP is secure under the RSA asssumption", in *Advances in Cryptology – Proceedings of CRYPTO '2001*, J. Kilian, Ed., *Lecture Notes in Computer Science*. Berlin: Springer, 2001, vol. 2139, pp. 260–274.

[32] O. Goldreich, S. Goldwasser, and S. Micali, "How to construct random functions", *J. ACM*, vol. 33, no. 4, pp. 792–807, 1986.

[33] S. Goldwasser and S. Micali, "Probabilistic encryption", *J. Comput. Syst. Sci.*, vol. 28, pp. 270–299, 1984.

[34] S. Goldwasser, S. Micali, and C. Rackoff, "The knowledge complexity of interactive proof systems", in *Proceedings of the 17th ACM Symposium on the Theory of Computing (STOC '85)*. New York: ACM Press, 1985, pp. 291–304.

[35] S. Goldwasser, S. Micali, and R. Rivest, "A "paradoxical" solution to the signature problem", in *Proceedings of the 25th Symposium on the Foundations of Computer Science (FOCS '84)*. New York: IEEE, 1984, pp. 441–448.

[36] S. Goldwasser, S. Micali, and R. Rivest, "A digital signature scheme secure against adaptative chosen-message attacks", *SIAM J. Comput.*, vol. 17, no. 2, pp. 281–308, 1988.

[37] C. Hall, I. Goldberg, and B. Schneier, "Reaction attacks against several public-key cryptosystems", in *Proceedings of the International Conference on Information and Communications Security 1999*, *Lecture Notes in Computer Science*. Springer, 1999, pp. 2–12.

[38] J. Håstad, "Solving simultaneous modular equations of low degree", *SIAM J. Comput.*, vol. 17, pp. 336–341, 1988.

[39] A. Joux and R. Lercier, "Improvements to the general Number Field Sieve for discrete logarithms in prime fields", *Math. Comput.*, 2000 (to appear).

[40] M. Joye, J. J. Quisquater, and M. Yung, "On the power of misbehaving adversaries and security analysis of the original EPOC", in *The Cryptographers' Track at RSA Conference '2001 (RSA '2001)*, D. Naccache, Ed., *Lectures Notes in Computer Science*. Berlin: Springer, 2001, vol. 2020, pp. 208–222.

[41] KCDSA Task Force Team. The Korean Certificate-based Digital Signature Algorithm. Submission to IEEE P1363a. Aug. 1998. [Online]. Available: http://grouper.ieee.org/groups/1363/.

[42] P. C. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems", in *Advances in Cryptology – Proceedings of CRYPTO '96*, N. Koblitz, Ed., *Lecture Notes in Computer Science*. Berlin: Springer, 1996, vol. 1109, pp. 104–113.

[43] P. C. Kocher, J. Jaffe, and B. Jun, "Differential power analysis", in *Advances in Cryptology – Proceedings of CRYPTO '99*, M. Wiener, Ed., *Lecture Notes in Computer Science*. Berlin: Springer, 1999, vol. 1666, pp. 388–397.

[44] A. Lenstra and H. Lenstra, *The Development of the Number Field Sieve*, *Lecture Notes in Mathematics*. Springer, 1993, vol. 1554.

[45] A. Lenstra and E. Verheul, "Selecting cryptographic key sizes", in *Workshop on Practice and Theory in Public-Key Cryptography (PKC '2000)*, H. Imai and Y. Zheng, Eds., *Lecture Notes in Computer Science*. Berlin: Springer, 2000, vol. 1751, pp. 446–465.

[46] A. K. Lenstra, H. W. Lenstra, and L. Lovász, "Factoring polynomials with rational coefficients", *Math. Ann.*, vol. 261, no. 4, pp. 515–534, 1982.

[47] H. Lenstra, "On the Chor-Rivest knapsack cryptosystem", *J. Crypt.*, vol. 3, pp. 149–155, 1991.

[48] J. Manger, "A chosen ciphertext attack on RSA optimal asymmetric encryption padding (OAEP) as standardized in PKCS $\#1$", in *Advances in Cryptology – Proceedings of CRYPTO '2001*, J. Kilian, Ed., *Lecture Notes in Computer Science*. Berlin: Springer, 2001, vol. 2139, pp. 230–238.

[49] R. J. McEliece, "A public-key cryptosystem based on algebraic coding theory". DSN Progress Rep., 42-44:114–116, 1978. Jet Propulsion Laboratories, CALTECH.

[50] G. Miller, "Riemann's hypothesis and tests for primality", *J. Comput. Syst. Sci.*, vol. 13, pp. 300–317, 1976.

[51] D. M'Raïhi, D. Naccache, D. Pointcheval, and S. Vaudenay, "Computational alternatives to random number generators", in *Fifth Annual Workshop on Selected Areas in Cryptography (SAC '98)*, *Lectures Notes in Computer Science*. Berlin: Springer, 1998, vol. 1556, pp. 72–80.

[52] M. Naor and M. Yung, "Public-key cryptosystems provably secure against chosen ciphertext attacks", in *Proceedings of the 22nd ACM Symposium on the Theory of Computing (STOC '90)*. New York: ACM Press, 1990, pp. 427–437.

[53] V. I. Nechaev, "Complexity of a determinate algorithm for the discrete logarithm", *Math. Not.*, vol. 55, no. 2, pp. 165–172, 1994.

[54] NIST. Digital Signature Standard (DSS). Federal Information Processing Standards PUBlication 186, Nov. 1994.

[55] NIST. Secure Hash Standard (SHS). Federal Information Processing Standards PUBlication 180–1, Apr. 1995.

[56] NIST. Secure Hash Algorithm 256/384/512. Oct. 2000.

[57] K. Ohta and T. Okamoto, "On concrete security treatment of signatures derived from identification", in *Advances in Cryptology – Proceedings of CRYPTO '98*, H. Krawczyk, Ed., *Lecture Notes in Computer Science*. Berlin: Springer, 1998, vol. 1462, pp. 354–369.

[58] T. Okamoto and D. Pointcheval, "REACT: rapid enhanced-security asymmetric cryptosystem transform", in *The Cryptographers' Track at RSA Conference '2001 (RSA '2001)*, D. Naccache, Ed., *Lectures Notes in Computer Science*. Berlin: Springer, 2001, vol. 2020, pp. 159–175.

[59] T. Okamoto and D. Pointcheval, "RSA-REACT: an alternative to RSA-OAEP", in *Second NESSIE Worksh.*, Sept. 2001.

[60] T. Okamoto and D. Pointcheval, "The gap-problems: a new class of problems for the security of cryptographic schemes", in *Workshop on Practice and Theory in Public-Key Cryptography (PKC '2001)*, K. Kim, Ed., *Lecture Notes in Computer Science*. Berlin: Springer, 2001, vol. 1992.

[61] D. Pointcheval, "Chosen-ciphertext security for any one-way cryptosystem", in *Workshop on Practice and Theory in Public-Key Cryptography (PKC '2000)*, H. Imai and Y. Zheng, Eds., *Lecture Notes in Computer Science*. Berlin: Springer, 2000, vol. 1751, pp. 129–146.

[62] D. Pointcheval and J. Stern, "Security proofs for signature schemes", in *Advances in Cryptology – Proceedings of EUROCRYPT '96*, U. Maurer, Ed., *Lecture Notes in Computer Science*. Berlin: Springer, 1996, vol. 1070, pp. 387–398.

[63] D. Pointcheval and J. Stern, "Security arguments for digital signatures and blind signatures", *J. Crypt.*, vol. 13, no. 3, pp. 361–396, 2000.

[64] D. Pointcheval and S. Vaudenay, "On provable security for digital signature algorithms". Technical Rep., Laboratoire d'Informatique de l'École Normale Supérieure, Oct. 1996.

[65] J. M. Pollard, "Monte Carlo methods for index computation (mod p)", *Math. Comput.*, vol. 32, no. 143, pp. 918–924, 1978.

[66] M. O. Rabin, "Digitalized signatures", in *Foundations of Secure Computation*, R. Lipton and R. D. Millo, Eds. New York: Academic Press, 1978, pp. 155–166.

[67] C. Rackoff and D. R. Simon, "Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack", in *Advances in Cryptology – Proceedings of CRYPTO '91*, J. Feigenbaum, Ed., *Lecture Notes in Computer Science*. Berlin: Springer, 1992, vol. 576, pp. 433–444.

[68] R. Rivest, "The MD5 message-digest algorithm". RFC 1321. The Internet Engineering Task Force, Apr. 1992.

[69] R. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public key cryptosystems", *Commun. ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[70] RSA Data Security, Inc. Public Key Cryptography Standards – PKCS. [Online]. Available: http://www.rsa.com/rsalabs/pubs/PKCS/.

[71] C. P. Schnorr, "Efficient identification and signatures for smart cards", in *Advances in Cryptology – Proceedings of CRYPTO '89*, G. Brassard, Ed., *Lecture Notes in Computer Science*. Berlin: Springer, 1990, vol. 435, pp. 235–251.

[72] C. P. Schnorr, "Efficient signature generation by smart cards", *J. Crypt.*, vol. 4, no. 3, pp. 161–174, 1991.

[73] C. P. Schnorr and M. Jakobsson, "Security of signed El Gamal encryption", in *Advances in Cryptology – Proceedings of ASIACRYPT '2000*, T. Okamoto, Ed., *Lecture Notes in Computer Science*. Berlin: Springer, 2000, vol. 1976, pp. 458–469.

[74] D. Shanks, "Class number, a theory of factorization, and Genera", in *Proceedings of the Symposium on Pure Mathematics*. AMS, 1971, vol. 20, pp. 415–440.

[75] C. E. Shannon, "Communication theory of secrecy systems", *Bell Syst. Tech. J.*, vol. 28, no. 4, pp. 656–715, 1949.

[76] H. Shimizu, "On the improvement of the Håstad bound", in *1996 IEICE Fall Conf.*, 1996, vol. A-162 (in Japanese).

[77] V. Shoup, "Lower bounds for discrete logarithms and related problems", in *Advances in Cryptology – Proceedings of EUROCRYPT '97*, W. Fumy, Ed., *Lecture Notes in Computer Science*. Berlin: Springer, 1997, vol. 1233, pp. 256–266.

[78] V. Shoup, "OAEP reconsidered", in *Advances in Cryptology – Proceedings of CRYPTO '2001*, J. Kilian, Ed., *Lecture Notes in Computer Science*. Berlin: Springer, 2001, vol. 2139, pp. 239–259.

[79] Y. Tsiounis and M. Yung, "On the security of El Gamal based encryption", in *Workshop on Practice and Theory in Public-Key Cryptography (PKC '98)*, H. Imai and Y. Zheng, Eds., *Lecture Notes in Computer Science*. Berlin: Springer, 1998.

[80] S. Vaudenay, "Cryptanalysis of the Chor-Rivest scheme", in *Advances in Cryptology – Proceedings of CRYPTO '98*, H. Krawczyk, Ed., *Lecture Notes in Computer Science*. Berlin: Springer, 1998, vol. 1462, pp. 243–256.

[81] G. S. Vernam, "Cipher printing telegraph systems for secret wire and radio telegraphic communications", *J. Am. Inst. Electr. Eng.*, vol. 45, pp. 109–115, 1926.

David Pointcheval is a Ph.D. in cryptology and a researcher at CNRS, in the computer science department at the École Normale Superieure in Paris, France. His research focuses on provable security, using reduction techniques. He has already published more than 40 papers in journals and international conferences on identification, signature, encryption and key agreement schemes. These proofs of security are not only theoretical results, but have practical impacts in evaluating the actual security of specific implementations according to the parameters. He therefore collaborates with several companies, all over the world, and is inventor of 7 patents.
e-mail: David.Pointcheval@ens.fr
LIENS – CNRS, École Normale Supérieure
45 rue d'Ulm, 75230 Paris Cedex 05, France

Paper

Analysis of cryptographic protocols using logics of belief: an overview

David Monniaux

Abstract — When designing a cryptographic protocol or explaining it, one often uses arguments such as "since this message was signed by machine *B*, machine *A* can be sure it came from *B*" in informal proofs justifying how the protocol works. Since it is, in such informal proofs, often easy to overlook an essential assumption, such as a trust relation or the belief that a message is not a replay from a previous session, it seems desirable to write such proofs in a formal system. While such logics do not replace the recent techniques of automatic proofs of safety properties, they help in pointing the weaknesses of the system. In this paper, we present briefly the BAN (Burrows-Abadi-Needham) formal system [10, 11] as well as some derivative. We show how to prove some properties of a simple protocol, as well as detecting undesirable assumptions. We then explain how the manual search for proofs can be made automatic. Finally, we explain how the lack of proper semantics can be a bit worrying.

Keywords — cryptographic protocols, logics of belief, BAN, GNY, decidability.

1. Why logics of belief?

Cryptographic protocols are usually specified as sequences of messages in the following kind of format:

Needham-Schroeder shared-keys protocol [10, 17, 24]

1. $P \to S \ : \ P, Q, N_p$
2. $S \to P \ : \ \left\{N_p, Q, K_{pq}, \{K_{pq}, P\}_{K_{qs}}\right\}_{K_{ps}}$
3. $P \to Q \ : \ \{K_{pq}, P\}_{K_{qs}}$
4. $Q \to P \ : \ \{N_q\}_{K_{pq}}$
5. $P \to Q \ : \ \{N_q - 1\}_{K_{pq}}$

where S, P, Q are machines or *principals*, or rather roles for machines in this protocol. S, as usual, designates a server.

N_p and N_q are *nonces* [21, §10.5]; these are random numbers (in this case, chosen respectively by P and Q) used to prevent *replay attacks*. Such attacks consist in an intruder replaying parts of messages recorded during previous sessions. The usual use of nonces is that the principals check that the values in certain encrypted message fields correspond to the correct values of the nonces for this session; discrepancies, arising from instance from messages recorded during previous sessions, get detected, preventing the principals from accepting those messages in a successful run of the protocol. From the point of view of protocol analysis, nonces are treated as being distinct from any other data used in the protocol. A related concept is that of *confounders* [21, §10.5], random numbers incorporated into messages to foil chosen plaintext attacks on public-key ciphers.

K_{xy} is a generic notation for a key shared between x and y. The goal of this protocol is to allow P and Q to agree on a shared communication key K_{pq}; for this, on the one hand, P and Q call a trusted server S which generates the key during the execution of the protocol; on the other hand, S communicates with P and Q using shared keys, K_{ps} and K_{qs} respectively, which are supposed to be known initially by the concerned parties.

The above description is a bit ambiguous, since it uses the same name (say, K) both for data that a principal generates by itself and for data that a principal receives from outside. For instance, in message 1, N_p is generated by P and thus treated by P as a known constant, but is received by S and thus treated by S as a variable. It can nevertheless be made unambiguous by distinguishing those two uses. From such an explicit description we can derive a semantics; that is, we describe in a mathematical way the actions of the principal. We also assume that we are in the Dolev-Yao model [13]: the cryptography is perfect, the intruder has full control of the network and can listen to, cancel and forge messages. Various analysis techniques, some of which considerably automated [5, 6, 8, 20, 26, 29, 32, and many others], have been applied to this model to obtain proofs of certain properties, and more particularly secrecy.

For all the successes of the Dolev-Yao model, using it to plan the design of a protocol is unnatural for a human. People do not design protocols by enumerating all the actions that could take place; they rather think of higher-order concepts such as "secret key only known to A and B and used to communicate between them" and form inferences such as "if a message arrives encrypted with a key known only to me and machine M, and I did not send it originally, then it must have been sent by M"[1]. Such reasoning is informal, which can be seen as a weakness. For this reason, some *logics of belief*, aiming at formalizing such inferences, have been proposed. The first of these was the so-called BAN logic from Burrows, Abadi and Needham [10, 11], which was followed by more expressive and elaborate extensions such as GNY (Gong, Needham and Yahalom [16, 17]), (Syverson and van Oorschot [33, 34]) and CKT5 [9]. One limitation of these logics is the need to annotate the protocols with logical assertions that are assumed to represent the intent of the sender of the message, as well as logical assumptions on the secrecy or freshness

[1] See [36] for a long discussion on such issues.

of certain pieces of information. Also, they cannot verify secrecy; in fact, *they make the implicit assumption that secrets are protected* [25].
BAN and subsequent logics are *modal logics of belief*; they deal with the beliefs that the principals can hold about their environment, for instance, about the distribution of the shared keys. That notion of "beliefs" is to be understood as the beliefs that a human playing the role of the principal may reasonably hold; "sensible" rules of deduction will be provided in the definition of the logic. As we will explain later (Section 4), it is difficult to provide a more precise semantics.

2. A short presentation of BAN and GNY logics

2.1. BAN logic

BAN logic [10, 11] is a many-sorted modal logic, which distinguishes between several sorts of objects: principals, encryption keys, nonces, and formulas, or statements. The first three sorts of objects have already been seen 1; the last sort is defined by the following syntax (taken from [10, pp. 4–5]; we left out two less used constructs):[2]

- $P \models X$: *P believes X*.
- $P \triangleleft X$: *P sees X*. *P* initially knew or has received the message *X* and can read and repeat it.
- $P |\!\sim X$: *P once said X*. *P* has at one time sent a message containing the statement *X*. It is not known whether the message was sent long ago or during the current session of the protocol, but it is known that *P* believed *X* when it sent the messages.
- $P \Rrightarrow X$: *P has jurisdiction* over *X* and should be trusted on this matter. For instance, key distribution servers will be trusted for statements pertaining to keys.
- $\sharp(X)$: *X* is *fresh*; that is, *X* has not been sent in a message at any time before the current run of the protocol. This is usually true for nonces; $\sharp(X)$ will then be used as a complement to $P |\!\sim X$ to establish that a message from *P* is really about the current session and is not some old recorded message used by the intruder in a replay attack.
- $P \overset{K}{\leftrightarrow} Q$: *P* and *Q* may use the shared key *K* to communicate. The key *K* is good, in that it will never be discovered by any principal except *P* or *Q*, or a principal trusted by either *P* or *Q*. Note that we make here the assumption that secrets are protected. This symbol is commutative, i.e. $P \overset{K}{\leftrightarrow} Q$ is equivalent to $Q \overset{K}{\leftrightarrow} P$.

[2]We use the original notation from the authors [10], who later preferred a more readable, albeit more verbose, notation [11]. We unfortunately cannot use this latter notation due to width constraints.

- $\overset{+K}{\mapsto} P$: *P* has *K* as a public key. The matching secret key (the inverse of *K*, denoted $-K$) will never be discovered by any principal except *P*, or a principal trusted by *P*.
- $\{X\}_K$: This represents the formula *X* encrypted under the key *K*. A weird point of BAN-like logics is that they consider that one can encrypt beliefs represented in formula. We shall now see why.

  Since messages are considered from the point of view of their meaning, a message K_{pq} conveying a key to be used between *P* and *Q* is represented in the logic as $P \overset{K_{pq}}{\longleftrightarrow} Q$. Since the key is generally encrypted so as to not being divulged to the intruder, the actually transmitted message is encrypted, for instance $\{K_{pq}\}_{K_{qs}}$. The corresponding formula is $\left\{P \overset{K_{pq}}{\longleftrightarrow} Q\right\}_{K_{qs}}$.
- (X,Y) represents the pair, or concatenation, of *X* and *Y*. Note that this symbol will be treated as commutative and associative.

We shall now see the deduction rules of BAN logic. A *deduction rule* is simply a set of *premises*, or *hypotheses* $\mathscr{H}_1, \ldots, \mathscr{H}_n$ and a *conclusion* $\mathscr{C}$ written as formulas with variables. Those variables stand for any formula, principal or nonce. We shall write such a rule as follows:

$$\frac{\mathscr{H}_1 \cdots \mathscr{H}_n}{\mathscr{C}}.$$

Such rules allow writing proofs as trees, whose leaves are the assumptions of the protocol or some already proved intermediary results and whose nodes are applications of the rules (see Figs. 1 and 2 for examples of somewhat complex proof trees). The notation

$$\frac{\begin{matrix}\vdots\ \alpha_1 \\ \mathscr{H}_1\end{matrix} \quad \cdots \quad \begin{matrix}\vdots\ \alpha_n \\ \mathscr{H}_n\end{matrix}}{\mathscr{C}}$$

means that α_i designates the branch of the proof tree whose root is $\mathscr{H}_i$. In our list of the rules for BAN logic, we shall use this notation to identify some premises in some rules, the use of which will be explained in Subsection 3.2.

- The *message-meaning* rules concern the interpretation of messages authenticated by encryption using a shared or private key:

$$\frac{\begin{matrix}\vdots\ p_1 \\ P \models P \overset{K}{\leftrightarrow} Q\end{matrix} \quad \begin{matrix}\vdots\ p_2 \\ P \triangleleft \{X\}_K\end{matrix}}{P \models Q |\!\sim X}\ \mathsf{MM1}$$

  The reasoning behind that rule is that if a key *K* is shared between two principals *P* and *Q* and is kept secret, if *P* sees a message encrypted with *K*, then it can assume it comes from Q. An additional (and easy to overlook) assumption is that the message should not have originally come from P. Burrows, Abadi and Needham justify this by explaining that $\{X\}_K$ is actually an abbreviation for $\{X\}_K$ *from* P, meaning that the encryption was done by P. It is assumed that each principal can recognize messages that it encrypted itself and ignore them. The message meaning rule can then be rewritten as:

$$\frac{P \mid\equiv P \overset{K}{\leftrightarrow} Q \qquad P \triangleleft \{X\}_K \ \mathit{from} \neq P}{P \mid\equiv Q \mid\!\sim X}\ \mathsf{MM1}.$$

This is a bit uneasy. GNY logic (Subsection 2.3) introduces a symbol $\star$, meaning *not originated here*, which makes such considerations internal to the logic:

$$\frac{\overset{\vdots\ p_1}{P \mid\equiv \overset{+K}{\mapsto} Q} \quad \overset{\vdots\ p_2}{P \triangleleft \{X\}_{-K}}}{P \mid\equiv Q \mid\!\sim X}\ \mathsf{MM2}.$$

- The *nonce verification* or *freshness* rule expresses the check that a message is recent (has been emitted in the current session) and thus that the sender still believes in it. The freshness condition is thus meant against replay attacks:

$$\frac{\overset{\vdots\ a}{P \mid\equiv \sharp(X)} \quad \overset{\vdots\ p}{P \mid\equiv Q \mid\!\sim X}}{P \mid\equiv Q \mid\equiv X}\ \mathsf{NV}.$$

- The *jurisdiction* rule states that if P believes that Q has jurisdiction over X then P trusts Q on the truth of X:

$$\frac{\overset{\vdots\ p}{P \mid\equiv Q \Rightarrow X} \quad \overset{\vdots\ a}{P \mid\equiv Q \mid\equiv X}}{P \mid\equiv X}\ \mathsf{J}.$$

- Unsurprisingly, a principal believes a group of statements if and only if it believes each one. We recall that pairs are treated as associative and commutative:

$$\frac{P \mid\equiv X \quad P \mid\equiv Y}{P \mid\equiv (X,Y)}\ \mathsf{BE1} \qquad \frac{\overset{\vdots\ p}{P \mid\equiv (X,Y)}}{P \mid\equiv X}\ \mathsf{BE2}$$

$$\frac{\overset{\vdots\ p}{P \mid\equiv Q \mid\equiv (X,Y)}}{P \mid\equiv Q \mid\equiv X}\ \mathsf{BE3}.$$

Other similar rules may be introduced if necessary, such as

$$\frac{P \mid\equiv Q \mid\equiv X \quad P \mid\equiv Q \mid\equiv Y}{P \mid\equiv Q \mid\equiv (X,Y)}\ \mathsf{BE4}.$$

- Similarly, if a principal said a group of things, it said each of them individually. Note that the converse is not true, since it would imply that the principal said all the things at the same moment:

$$\frac{\overset{\vdots\ p}{P \mid\equiv Q \mid\!\sim (X,Y)}}{P \mid\equiv Q \mid\!\sim X}\ \mathsf{SG}.$$

- If a principal sees a formula, then he also sees its components, provided he knows the necessary keys:

$$\frac{\overset{\vdots\ p}{P \triangleleft (X,Y)}}{P \triangleleft X}\ \mathsf{SP1} \qquad \frac{\overset{\vdots\ p_1}{P \triangleleft \{X\}_K} \quad \overset{\vdots\ p_2}{P \mid\equiv P \overset{K}{\leftrightarrow} Q}}{P \triangleleft X}\ \mathsf{SP2}.$$

Note that the hypothesis is $P \mid\equiv P \overset{K}{\leftrightarrow} Q$, not $P \triangleleft K$, which would seem logical. In fact, this rule could perhaps be replaced by the following pair of rules:

$$\frac{\overset{\vdots\ p}{P \triangleleft \{X\}_K} \quad \overset{\vdots\ a}{P \triangleleft K}}{P \triangleleft X} \qquad \frac{\overset{\vdots\ p}{P \mid\equiv P \overset{K}{\leftrightarrow} Q}}{P \triangleleft K}.$$

The usual rule for public-key cryptosystems is that message encrypted with the public keys are decipherable using the private key:

$$\frac{\overset{\vdots\ p}{P \triangleleft \{X\}_{+K}} \quad \overset{\vdots\ a}{P \mid\equiv \overset{+K}{\mapsto} P}}{P \triangleleft X}\ \mathsf{SP3}.$$

Note that this last rule supposes that if P believes that K is its public key, then it holds the corresponding private key.

The following optional rule expresses the fact that for certain public-key cryptosystems (like RSA [21, 28, 30]), it is possible for anybody with the public key to decipher a message encrypted with the private key:

$$\frac{\overset{\vdots\ p}{P \triangleleft \{X\}_{-K}} \quad \overset{\vdots\ a}{P \mid\equiv \overset{+K}{\mapsto} Q}}{P \triangleleft X}\ \mathsf{SP4}.$$

- If one part of a formula is known to be fresh, then the entire formula must also be fresh:

$$\frac{P \models \sharp(X)}{P \models \sharp(X,Y)}\ \mathsf{FR1} \qquad \frac{P \models \sharp(X)}{P \models \sharp(\{X\}_K)}\ \mathsf{FR2}.$$

An important point about BAN logic is that it was intended to be a starting point for logics adapted for certain particular uses. A person aiming at applying such techniques to protocols may have to introduce additional constructs and rules to reflect the particularities of the system. The use of automatic decision procedures (see Section 3) may help in this respect to identify the missing rules and assumptions – which sometimes are indeed assumptions about the system that the designer had not noticed.

2.2. The Needham-Schroeder protocol in BAN logic

We shall see here how to formalize and analyze the Needham-Schroeder shared-keys protocol (see Section 1). This protocol is of particular importance since many others, such as Kerberos [22], have been derived from it; furthermore, it has a serious weakness, an undesirable assumption, which can be demonstrated in the logical analysis.

We shall follow the analysis in [10, § 5]. The first step is to convert the protocol description into a sequence of BAN assumptions. Each line of the form $A \to B : F$ induces a formula of the form $B \triangleleft F$. We remove indications that play no role in the logical deductions, such as the names of the principals, and we replace some elements by their semantic meaning: the freshly generated key K_{pq}, meant to be used between P and Q, is idealized as the pair of formulas $P \overset{K_{pq}}{\longleftrightarrow} Q$ and $\sharp\left(P \overset{K_{pq}}{\longleftrightarrow} Q\right)$:

$$\begin{array}{lcl}
2.\ S \to P & : & \left\{N_p, P \overset{K_{pq}}{\longleftrightarrow} Q, \sharp\left(P \overset{K_{pq}}{\longleftrightarrow} Q\right), \left\{P \overset{K_{pq}}{\longleftrightarrow} Q\right\}_{K_{qs}}\right\}_{K_{ps}} \\
3.\ P \to Q & : & \left\{P \overset{K_{pq}}{\longleftrightarrow} Q\right\}_{K_{qs}} \\
4.\ Q \to P & : & \left\{N_q, P \overset{K_{pq}}{\longleftrightarrow} Q\right\}_{K_{pq}} \mathit{from}\ Q \\
5.\ P \to Q & : & \left\{N_q, P \overset{K_{pq}}{\longleftrightarrow} Q\right\}_{K_{pq}} \mathit{from}\ P.
\end{array}$$

The first message is omitted, since it does not contribute to the logical properties of the protocol. We should nevertheless not forget that N_p is created just before this first message is sent and thus is assumed to be fresh. The case of the last two messages is more interesting. In the concrete protocol, $N_q - 1$ is used instead of N_q in message 5 so that messages 4 and 5 are different. An intruder cannot replay to P its own message 4. We therefore make this impossibility explicit by using the construction $\left\{N_q, P \overset{K_{pq}}{\longleftrightarrow} Q\right\}_{K_{pq}}$ $\mathit{from}\ P$, give in the above explanation of the message-meaning rule.

To start, we give some assumptions:

- The first assumptions state that the principals know how to communicate using shared-key cryptography between the clients and the server:

$$\begin{array}{ll}
P \models P \overset{K_{ps}}{\longleftrightarrow} S & Q \models Q \overset{K_{qs}}{\longleftrightarrow} S \\
S \models P \overset{K_{ps}}{\longleftrightarrow} S & S \models Q \overset{K_{qs}}{\longleftrightarrow} S \\
S \models P \overset{K_{pq}}{\longleftrightarrow} Q. &
\end{array}$$

- P and Q trust the server in producing a fresh and correct shared key. They will accept whatever key K that the server will supply; we shall therefore specify these assumptions as *axiom schemes*, where K can be instanced by any value:

$$\begin{array}{l}
\forall K\ P \models S \mapsto P \overset{K}{\leftrightarrow} Q, \quad P \models S \mapsto \sharp\left(P \overset{K}{\leftrightarrow} Q\right) \\
\forall K\ Q \models S \mapsto P \overset{K}{\leftrightarrow} Q.
\end{array}$$

  Since the only value for K that makes sense to reach useful conclusions is K_{pq}, we can replace these axiom schemes by axioms:

$$\begin{array}{ll}
P \models S \mapsto P \overset{K_{pq}}{\longleftrightarrow} Q & P \models S \mapsto \sharp(P \overset{K_{pq}}{\longleftrightarrow} Q) \\
Q \models S \mapsto P \overset{K_{pq}}{\longleftrightarrow} Q. &
\end{array}$$

  This is also required for our automatic proof technique (Section 3).

- Unsurprisingly, each principal believes in the freshness of what it generates:

$$\begin{array}{ll}
P \models \sharp(N_p) & Q \models \sharp(N_q) \\
S \models \sharp(P \overset{K_{pq}}{\longleftrightarrow} Q). &
\end{array}$$

- This last assumption is needed to reach the protocol goals, but is wrong. As pointed out in [10]:

  > [...] the protocol has been criticized for using this assumption, and the authors did not realize they were making it.

  We shall discuss below the unwanted consequences of this assumption:

$$\forall K\ Q \models \sharp(P \overset{K}{\leftrightarrow} Q). \tag{1}$$

Let us now see the proofs using BAN logic. First, principal P has to ensure that the key is fresh (Fig. 1). It is then possible to derive $P \models P \overset{K_{pq}}{\longleftrightarrow} Q$ (Fig. 2).

Also, $P \triangleleft \left\{P \overset{K_{pq}}{\longleftrightarrow} Q\right\}_{K_{qs}}$. Since P has seen that part of the message, it can retransmit it to Q. At this point, Q decrypts the message, and we obtain:

$$\frac{Q \models Q \overset{K_{qs}}{\longleftrightarrow} S \quad Q \triangleleft \left\{P \overset{K_{pq}}{\longleftrightarrow} Q\right\}_{K_{qs}}}{Q \models S \mid\!\sim P \overset{K_{pq}}{\longleftrightarrow} Q}\ \mathsf{MM1}.$$

Let us note that Q has no means to check that this message is fresh except for assumption 1. If we make this assumption, we get

$$\dfrac{\dfrac{Q\mid\equiv S\mid\sim P\xleftrightarrow{K_{pq}}Q \qquad \dfrac{\forall K\ Q\mid\equiv\sharp(P\xleftrightarrow{K}Q)}{Q\mid\equiv\sharp(P\xleftrightarrow{K_{pq}}Q)}}{Q\mid\equiv S\mid\equiv P\xleftrightarrow{K_{pq}}Q}\,\text{NV} \qquad \dfrac{\forall K\ Q\mid\equiv S\Rightarrow P\xleftrightarrow{K}Q}{Q\mid\equiv S\Rightarrow P\xleftrightarrow{K_{pq}}Q}}{Q\mid\equiv P\xleftrightarrow{K_{pq}}Q}\,\text{J}.$$

The last two messages are for P and Q to be sure that the other one has indeed received the key and is ready to use it:

$$\dfrac{P\mid\equiv\sharp(P\xleftrightarrow{K_{pq}}Q) \qquad \dfrac{\dfrac{P\mid\equiv P\xleftrightarrow{K_{pq}}Q \quad P\triangleleft\left\{N_q,P\xleftrightarrow{K_{pq}}Q\right\}_{K_{pq}}}{P\mid\equiv Q\mid\sim N_q,P\xleftrightarrow{K_{pq}}Q}\,\text{MM1}}{P\mid\equiv Q\mid\sim P\xleftrightarrow{K_{pq}}Q}\,\text{SG}}{P\mid\equiv Q\mid\equiv P\xleftrightarrow{K_{pq}}Q}\,\text{NV}$$

$$\dfrac{\dfrac{\dfrac{Q\mid\equiv\sharp(N_q)}{Q\mid\equiv\sharp(N_q,P\xleftrightarrow{K_{pq}}Q)}\,\text{FR1} \qquad \dfrac{Q\mid\equiv P\xleftrightarrow{K_{pq}}Q \quad Q\triangleleft\left\{N_q,P\xleftrightarrow{K_{pq}}Q\right\}_{K_{pq}}}{Q\mid\equiv P\mid\sim N_q,P\xleftrightarrow{K_{pq}}Q}\,\text{MM1}}{Q\mid\equiv P\mid\equiv N_q,P\xleftrightarrow{K_{pq}}Q}\,\text{NV}}{Q\mid\equiv P\mid\equiv P\xleftrightarrow{K_{pq}}Q}\,\text{BE3}$$

In other words, each principal P or Q trusts the other one in believing they share a secret key K_{pq}.
Let us now discuss the weakness of the protocol: assumption 1 $\left(\forall K\ Q\mid\equiv\sharp(P\overset{K}{\leftrightarrow}Q)\right)$. It means that Q will accept a proposal for a key K_{pq} without being able to check whether this key is appropriate for this session. In fact, let us suppose that an intruder I has listened to the network and recorded a session involving P and Q. It therefore has recorded a valid message $\{K_{pq},P\}_{K_{qs}}$. Let us additionally assume that the intruder has managed to get hold of K_{pq}. Now the intruder impersonates P to initiate a run of the protocol with Q using that recorded information:

$$\begin{aligned}
3.\ & I\rightarrow Q\ :\ \{K_{pq},P\}_{K_{qs}}\\
4.\ & Q\rightarrow I\ :\ \{N_q\}_{K_{pq}}\\
5.\ & I\rightarrow Q\ :\ \{N_q-1\}_{K_{pq}}\ .
\end{aligned}$$

Now Q believes it can communicate with P using K_{pq}. Q will in good faith start talking with the intruder, believing the intruder is P. In other words, if one session key has been compromised, all subsequent sessions can be compromised as well. This contradicts one of the very motivations for the use of session keys which is "to limit exposure, with respect to both time period and quantity of data, in the event of (session) key compromise" [21, §12.2.2]. As [10] points out:

> Denning and Sacco pointed out that compromise of a session key can have very bad results: an intruder has unlimited time to find an old session key and to reuse it as though it were fresh (1981). Bauer, Berson, and Feiertag pointed out that there are even more drastic consequences if [P]'s private key is compromised: an intruder can use [P]'s key to obtain session keys to talk to many other principals, and can continue to use these session keys even after [P]'s key has been changed (1983). It is comforting that the logical analysis makes explicit the assumption.

BAN logic has thus been successful in identifying an unwanted assumption of a protocol on the freshness of a message, indicating the possibility of a replay attack.

2.3. A simple example in GNY logic

BAN logic was much criticized, on the one hand for being some kind of dubious idealization of the already idealized Dolev-Yao model, on the other hand for making unwanted assumptions. We have already seen the uneasy treatment that BAN makes of the situation where a principal P is sent a message $\{M\}_{K_{pq}}$, where K_{pq} is a shared key for P and Q: P can believe that this message originated from Q only if P is sure that this message is not a replay of one of its own messages. GNY logic is a BAN-like logic that solves this issue as well as others [17]:

> Our new approach seems to offer important advantages over the BAN approach. It does not require several universal assumptions which the BAN work does. For example, it does not assume that redundancy is always present in encrypted messages incorporating instead a new notion of recognizability which captures a recipient's expectation of the contents of messages he receives. Also, it does not assume that a principal can always determine whether a message was not once originated by himself.

GNY logic also separates what a principal says, what it believes and what it possesses. Other logics have been proposed to alleviate some other weaknesses of BAN logic [33, 34]. We shall restrict ourselves here to a cursory glance at GNY logic [17].
We shall consider a very simple (and admittedly silly) protocol:

$$\begin{aligned}
1.\ & A\rightarrow B\ :\ N_a\\
2.\ & B\rightarrow A\ :\ \{N_a\}_{-K_b}\\
3.\ & A\rightarrow B\ :\ \{K_{ab}\}_{+K_b}\ .
\end{aligned}$$

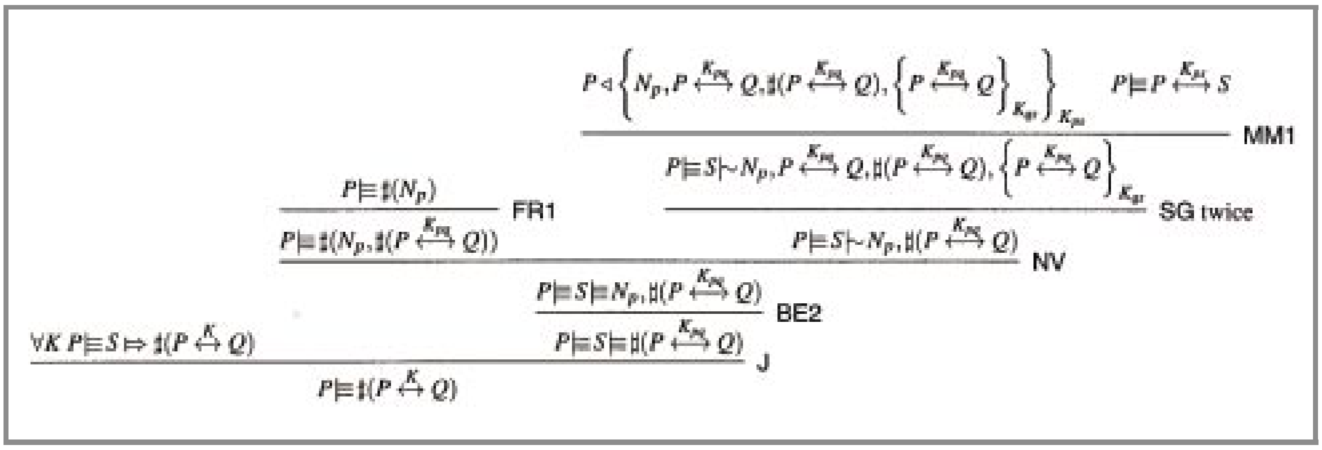


Fig. 1. Deriving freshness in BAN logic.

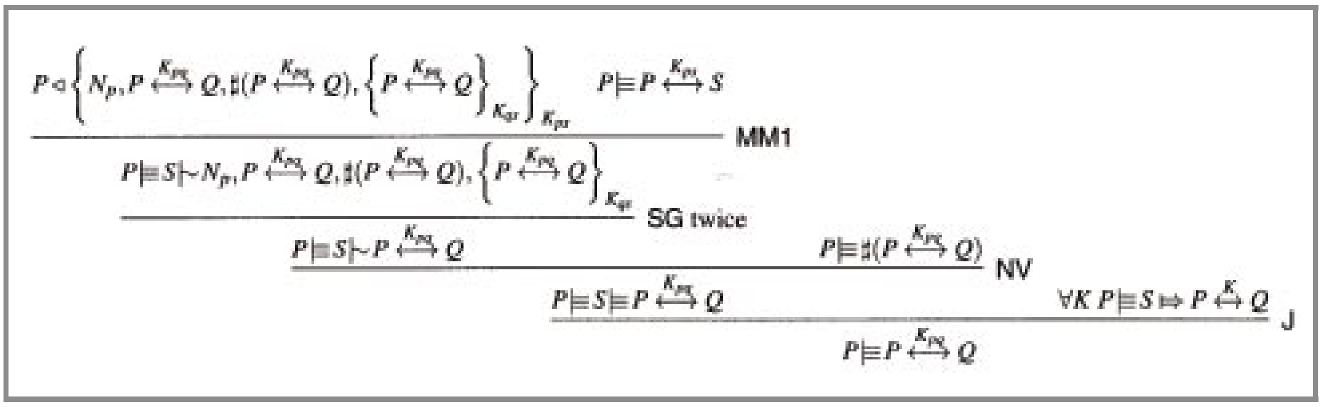


Fig. 2. Deriving that a key is shared in BAN logic.

This is to be understood as: in step 1, A sends a newly generated number N_a to B; B answers with the encryption of N_a by its private key $-K_b$; A answers with the encryption of a newly generated session key K_{ab} with B's public key $+K_b$.

To illustrate how belief-logic deductions work, we first show the idealized version of the protocol in GNY:

1. $B \triangleleft N_a$
2. $A \triangleleft \star \{N_a\}_{-K_b}$
3. $B \triangleleft \star \left(\{K_{ab}\}_{+K_b} \rightsquigarrow A \overset{K_{ab}}{\longleftrightarrow} B \right)$.

The star means that the following term was not originated by the party who receives it. The statement after the wavy arrow in 3 is an annotation meaning that K_{ab} is intended to be a shared secret key for use between A and B.

We also need some assumptions, written as follows in GNY:

(a) $A \ni +K_b$ (A possesses $+K_b$)

(b) $A \models \overset{+B}{\mapsto} +K_b$ (A believes that $+K_b$ is B's public key)

(c) $A \models \phi(N_a)$ (A believes N_a to be recognizable; that is, if A sees a message field that is supposed to be N_a, A can check whether it is or not)

(d) $A \models \sharp(N_a)$ (A believes N_a to be fresh; that is, to have been used for the first time in this run of the protocol).

See [17] for a complete list of the GNY inference rules and their designations. Using those rules, we can derive the conclusion $A \models B \ni N_a$ as in Fig. 3.

This means that from the fact, coming from protocol step 2, that A sees the message consisting of the encryption of N_a by the private key for the public/private couple of keys K_b, from assumption a (A possesses the corresponding public key), from assumption b (B uses the private key of the couple K_b) and from assumption c (A believes that it can recognize N_a), we deduce using rule I4 that A is entitled to believe that B once said N_a. Then, using assumption d, which is that N_a is fresh (has never been used in another session before), we deduce that A is entitled to believe that B possesses N_a.

The final goal of the protocol might be to cause B to believe that A believes that K_{ab} is the shared key ($A \models A \overset{K_{ab}}{\longleftrightarrow} B$). However, message 3 could have been forged by any intruder possessing $+K_b$, which is realistic since it is a public key,

$$\dfrac{\dfrac{A \lhd \star\{N_a\}_{-K_b} \quad A \ni +K_b \quad A \models \overset{+K_b}{\mapsto} +K_b \quad A \models \phi(N_a)}{A \models B \mid\sim N_a}\ \text{I4} \qquad A \models \sharp(N_a)}{A \models B \ni N_a}\ \text{I6}$$

Fig. 3. A derivation in GNY logic.

replacing K_{ab} by any key of his choice. The logic (correctly) fails to conclude that the protocol accomplishes this goal: this goal has no derivation in GNY logic from the above set of hypotheses.

3. Decidability

A little known fact about the modal logics of belief (at least BAN and GNY) is that they are *decidable* [23]. That is, there exists an algorithm that, given a finite set of hypotheses H_1 to H_n and a purported conclusion C, answers whether or not C follows from $H_1, \dots, H_n$. We shall see in this section the difficulties of establishing this property and a practical algorithm.

3.1. Position of the problem

Let us first remark that not all logics are decidable. For instance, set theory, the basis of usual mathematics, is *undecidable* [12]: that is, there exists no algorithm that takes as input a mathematical proposition and answers whether it is true or false. Furthermore, the analysis of cryptographic protocols in the Dolev-Yao model, given some very reasonable hypotheses, is also undecidable if an unbounded number of sessions is allowed, even with messages of bounded depth [14] [3].

There are two traditional methods to test whether a formula t admits a derivation from a set of hypotheses Γ in a rule system $\vdash$ (which we note by $\Gamma \vdash t$):

- **Forward chaining**, that is starting from the hypotheses Γ, apply all the possible deduction rules to deduce new formulas, then start again with the union of the hypotheses and the new formulas, until the formula t is discovered.

- **Backward chaining**, that is, starting from the purported conclusion, find all the rules and all the instantiations of the variables in them that yield that conclusion, then try recursively to prove the hypotheses of each of these rules with each of the instantiations; this requires backtracking.

There are two problems with the rule systems like BAN or GNY:

- Both forward chaining and backward chaining may fail to terminate.

- There are rules that are unsuitable for forward-chaining and rules that are unsuitable for forward-chaining. For instance,

$$\frac{P \models \sharp(X)}{P \models \sharp(X,Y)}\ \mathsf{FR1}$$

has a conclusion in which there are variables that are not found in the hypotheses. It is therefore impossible to apply forward chaining, except by introducing variables representing unknown formulas. Similarly, rule BE2 is not suitable for backward-chaining.

If we straightforwardly (and naively) implement the rules of BAN or GNY logic in a general-purpose automatic theorem prover, as it has been done [31], the prover is likely to search an infinite space of possible proofs, which means that the system does not terminate when the conclusion is not provable. Furthermore, even in cases of termination, the computation time might be prohibitive, because the search procedure explores many useless avenues.

The approach taken by Kindred and Wing [19] and generalized by ourselves [23] is a refinement on a combination of forward and backward chaining. We shall expose here briefly our method. This method analyses the GNY logic [17], but is generic enough to be applied to most similar logics. It is based on a careful application of forward-chaining.

3.2. Composition and decomposition rules

Let us take a look at BAN logic[4]. We consider a partition of the rules between these two classes:[5]

- *Decomposition rules*, in which all the variables of the conclusion are found in the premises (these rules are suitable for forward-chaining); for these rules, we distinguish the *principal premises* (which can be one or more) and the optional *auxiliary premises*; the

[3]See also Comon and Shmatikov's paper in this volume.

[4]We do similar work for a variant of GNY logic equivalent to GNY logic in [23].

[5]This partition of the set of rules into two classes is very similar to that of [19], where they are called respectively *growing* and *shrinking rules*. This is also similar to the introduction and elimination rules of natural deduction; see [15, p. 75] and [27 § II.1]. Our theorem on normal derivations is thus similar to the normalization theorem of natural deduction [27, § IV.1] or Gentzen's *Hauptsatz* [15, p. 105].

variables in the auxiliary premises are a subset of these in the principal premises; those rules in BAN are the ones listed above as

$$\dfrac{\overset{\vdots\; p_1}{\mathscr{H}_1} \quad \cdots \quad \overset{\vdots\; p_n}{\mathscr{H}_n} \quad \overset{\vdots\; a_1}{\mathscr{H}_{n+1}} \quad \cdots \quad \overset{\vdots\; a_m}{\mathscr{H}_{n+m}}}{\mathscr{C}}$$

- *Composition rules*, in which all the variables of the premises are found in the conclusion (these rules are suitable for backward-chaining).

The intuition is that composition rules introduce constructors, like a pair, and decomposition rules break these constructors, as in taking the first projection of a pair.

GNY and similar logics fulfill the *normal derivation criterion*: if there exists a derivation of $\Gamma \vdash t$ then there must also exist a *normal derivation* of $\Gamma \vdash t$. A derivation Δ of a conclusion $\Gamma \vdash t$ is said to be *normal* if there is no composition rule to be used as the root rule of the sub-derivation for a principal premise of a decomposition rule: for any decomposition rule d used in Δ:

$$\dfrac{\dfrac{\vdots}{\mathscr{P}_1}\,r_1 \quad \cdots \quad \dfrac{\vdots}{\mathscr{P}_n}\,r_n \quad \overset{\vdots}{\mathscr{A}_1} \quad \cdots \quad \overset{\vdots}{\mathscr{A}_m}}{C}\,d$$

where $\mathscr{P}_i$ are the principal hypotheses and $\mathscr{A}_j$ the auxiliary hypotheses, none of the rules $r_1, \ldots, r_n$ is a composition rule. In the opposite case, we say that there is a *detour* at r.

Informally, that means that it must be possible to derive anything that is derivable without having to compose something and decompose it afterwards;[6] for instance, in

$$\dfrac{\dfrac{\overset{\vdots\;\alpha}{P \models X} \quad \overset{\vdots\;\beta}{P \models Y}}{P \models (X,Y)}\,\text{BE1}}{P \models X}\,\text{BE2}$$

we compose a pair just to decompose it afterward, and we could have reached the same conclusion directly using only the α branch of the proof:

$$\overset{\vdots\;\alpha}{P \models X}\,.$$

3.3. Magic-set transformation

As we pointed out earlier, it is possible to implement straightforwardly BAN or GNY logic into forward-chaining or backward-chaining (Prolog-like) theorem provers; this is nevertheless clumsy since:

[6]In terms of natural deduction and similar systems, this is often called the *inversion principle* [27, ch. II].

- It is necessary to allow non-ground formulas, that is, formulas containing variables representing unknown sub-formulas. This means that the prover needs apply unification and not just pattern-matching when applying rules. This makes the implementation far more complex.
- Many unnecessary conclusions or intermediary goals are generated. In the case of BAN or GNY logic, this leads to non-termination when the purported conclusion is not reachable from the hypotheses.

Similar problems were encountered in the field of logic programming and queries in logical databases and led to the introduction of the *magic-set* transformation [7]. This transformation turns backward-chaining rules into forward-chaining ones, by generating only "relevant" sub-goals.[7] These sub-goals constitute the so-called "magic sets" associated with the predicates. The basic idea is that we should only begin investigating sub-goals of a conclusion only if some sub-goals have already been proved and some variables instanciated (thus the vocabulary of *Sideways Information Passing Strategy* or *SIPS*).

In our case, the partitions into composition and decomposition rules, and in that latter class, between principal and auxiliary premises gives us the SIPS. Furthermore, we choose our partition so that all the forward-chaining rules generated by the transformation make some measure decrease, thus ensuring termination. We then define another logic, introducing a special symbol, *goal*. $\boxdot X$ means that "we would like to compose X". Other authors call this symbol "magic".

Our transformation turns the composition rule

$$\dfrac{\mathscr{H}_1 \cdots \mathscr{H}_n}{\mathscr{C}}$$

into a pair

$$\dfrac{\boxdot\mathscr{C} \quad \mathscr{H}_1 \cdots \mathscr{H}_n}{\mathscr{C}} \qquad \dfrac{\boxdot\mathscr{C}}{\boxdot\mathscr{H}_1 \cdots \boxdot\mathscr{H}_n}$$

and adds to the decomposition rule

$$\dfrac{\mathscr{P}_1 \cdots \mathscr{P}_n \quad \mathscr{A}_1 \cdots \mathscr{A}_m}{\mathscr{C}},$$

where the one or more $\mathscr{P}_i$ are principal premises and the zero or more $\mathscr{A}_i$ are auxiliary premises, the triggering rule

$$\dfrac{\mathscr{P}_1 \cdots \mathscr{P}_m}{\boxdot\mathscr{A}_1 \cdots \boxdot\mathscr{A}_n}.$$

It can be proved [23] that

Theorem 1: For any set of hypotheses $\mathscr{H}$ and purported conclusion $\mathscr{C}$ for the $\vdash$ proof system,

$$\mathscr{H} \vdash \mathscr{C} \iff \mathscr{H}, \boxdot\mathscr{C} \vdash' \mathscr{C}.$$

[7]As it has been pointed out [7, § II], "relevant" means "[that might] be essential to the establishment of a fact that is in the answer". "Relevant" goals may actually be irrelevant to the query; the only insurance that we have is that for any provable fact there will be one proof of that fact for which we shall obtain all sub-goals as "relevant".

3.4. The decision procedure

The interest of turning the original problem on $\vdash$ into a problem on $\vdash'$ is twofold:

1. All rules in $\vdash'$ are suitable for forward-chaining.
2. For any finite set of hypotheses $\mathscr{H}$, the length of the derivations of the $\vdash'$-proofs starting from $\mathscr{H}$ is bounded. This condition is proved by giving a *weight function* assigning an integer weight $|F|$ to each formula F so that the weight of the conclusion of a $\vdash'$-rule is strictly less than the maximal weight of the premises[8] (Table 1).

To some extent, the existence of the weight function is intuitive: some rules are evidently composition rules (building a pair, for instance) or decomposition rules (splitting a pair), and the intuition is that the conclusion of a decomposition rule is "smaller" than the premises, and that the conclusion of a composition rule is "bigger" than the premises. The additional condition that the auxiliary premises are smaller than the principal ones is also intuitive, since the auxiliary premises often represent keys and the principal premises represent encrypted data containing the key as a sub-term. The rules resulting from the magic-set transformation then have conclusions that are "smaller" than their premises.

Table 1
The weight function for BAN logic

F	$\|F\|$
K	1
P	1
N_p	1
$P \mid\equiv X$	$1+\|X\|$
$P \triangleleft X$	$2+\|X\|$
$P \mid\sim X$	$3+\|X\|$
$P \Rightarrow X$	$3+\|X\|$
$\sharp(X)$	$1+\|X\|$
$P \overset{K}{\leftrightarrow} Q$	$2+\|K\|$
$\overset{+K}{\mapsto} P$	$2+\|K\|$
$\{X\}_K$	$3+\|X\|+\|K\|$
(X,Y)	$1+\|X\|+\|Y\|$

For any finite set $\mathscr{H}$ of hypotheses, the set of conclusions that can be derived from $\mathscr{H}$ is finite and can be enumerated by exhaustively applying the rules of $\vdash'$; this is often referred to as the *saturation* of the hypotheses by the forward-chaining system $\vdash'$. The decision procedure for $\vdash'$ is thus simple: to test whether $A \vdash' B$, it suffices to saturate A by $\vdash'$ and test whether B belongs to the set. Let us note that although $\mathscr{W}$ is used to prove the termination of the saturation process, that process does not need to compute $\mathscr{W}$.

[8]The problem is slightly more complex for GNY logic because of its *jurisdiction rule* [23].

Using Theorem 1, we obtain a decision procedure for $\vdash$ (which can be BAN logic or a modified version of GNY logic [23]).

Minimal care must be taken when implementing the saturation procedure, especially for more complex logics such as GNY. Naive implementations may lead to prohibitive costs. For instance, trying all possible rules in the fashion that to try a n-ary rule you match it against all the n-tuples of already derived formulas, until it ends, leads to prohibitive costs (in the case of GNY, $n = 6$, which makes the number of matchings grow in D^7, where D is the number of derivable formulas).

A first optimization we tried was based on the fact that one need not test all possible n-tuples, but only ones containing at least one "new" formula; that is, a formula made during the last application of the rules. This is not sufficient, since it reduces only to D^6; experimentally, this is far too slow.

Our implementation is based on the fact that to instantiate all the variables in a rule, you need not consider all the hypotheses; especially, in the modified versions of the composition rules, only one "goal" hypothesis suffices; in the decomposition and trigger rules, only the principal premises are needed. Expensive exhaustive searches for the fully instantiated hypotheses are replaced by a much faster binary search. On problems taken from the protocol literature, this implementation performed within seconds.[9] Implementations based on efficient general-purpose forward chaining systems are likely to perform even better.

Our goal is not only to "prove" protocols in the logic, but also to identify undesirable assumptions. Our analyzer can also help in that regard, since it generates a list of possibly desirable assumptions (the formulas F so that $\boxdot F$ is in the saturation of the problem by $\vdash'$, while F is not). Experimentally, the list given by the analyzer tends to contain the missing assumptions, but also many ludicrous ones. Heuristics may help to produce meaningful output to the protocol designer.

4. Semantics

We have so far defined a system of rules. But are we sure that they are the right rules? Are we sure we are not going to deduce something wrong because of a loophole in the system? It would be much better if we had a way of representing the concepts that are embodied in the formulas and check whether the deductive relationships expressed by the rules are actually true. For this, we must define the *semantics* of formulas in terms of *models*.

We shall remind the reader briefly of the semantic aspects of axiomatic methods. Let us take a simple example. It is possible to write proofs of facts in planar geometry using a few structural rules (*modus ponens* and other rules for basic logical connectors) as well as a few axioms on geometrical properties. This is the usual geometry that children learn at school. On the other hand, it is possible to

[9]The implementation is freely available from the author.

build a theory of geometry based on set theory, the integer, the rationals, the real field and Euclidean spaces. What is the relationship between those two theories? Every object (point, line …) of usual planar geometry can be represented by an object built from set theory. The same holds for the relationships between those objects (parallelism, intersections …) and even whole statements. These representations constitute a *model*. We say that a model $\mathscr{M}$ represents a formula F (noted $\mathscr{M} \models F$) if and only if the representation of the formula F is true in the model $\mathscr{M}$. A statement is said to be *valid* if it is true in all models. There are then two problems to consider:

- Is the system of rules that we consider *sound*? That is, are all deducible statements true in all models?
- Is the system of rules complete? That is, is there a proof for every valid statement that can be written in the system?

It is interesting to note that indeed it is possible to give a sound and complete axiomatic system for planar geometry [37] and quite a few other interesting theories.
Early attempts at giving semantics for logics of belief for cryptographic protocols gave only somehow "trivial" semantics: they basically said that what a principal believes is what it had come to believe following the rules. Such a semantics does not shed any light on what "belief" means in the context of cryptographic protocols. It seems desirable to have logics of belief proved to be sound with respect to a non-trivial semantics for beliefs, which will involve a notion of *possible worlds* [18]. Improved belief logics, proved to be sound with respect to possible world semantics, were therefore proposed [4, 34]. Later, semantics based on *strand spaces* were also proposed [35].

5. Conclusions

BAN, and similar logics, are useful to get an idea of the assumptions underlying the design of a cryptographic protocol. Their handling can be (partially) automated. While they do not provide the same sort of assurance as analyses in the Dolev-Yao model [13] or the spi-calculus and its variants [1–3], they can point mistakes in the design of protocols, including misplaced trust or failure to prevent replay attacks.

Acknowledgements

We thank J. Goubault-Larrecq, Jon Millen, and Luca Vigano for their many suggested improvements to the paper.

References

[1] M. Abadi and A. D. Gordon, "A calculus for cryptographic protocols: the spi calculus", in *ACM Conf. Comput. Commun. Secur.*, 1997, pp. 36–47.

[2] M. Abadi and A. D. Gordon, "Reasoning about cryptographic protocols in the spi calculus", in *CONCUR'97, Concurrency Theory, 8th International Conference*, A. Mazurkiewicz and J. Winkowski, Eds., *Lecture Notes in Computer Science.* Springer, 1997, vol. 1243, pp. 59–73.

[3] M. Abadi and A. D. Gordon, "A calculus for cryptographic protocols: the spi calculus". Res. Rep. 149, Compaq Systems Research Center, Palo Alto, CA, USA, Jan. 1998.

[4] M. Abadi and M. R. Tuttle, "A semantic for a logic of authentication", in *10th Annual ACM Symposium on Principles of Distributed Computing,* L. Logrippo, Ed. ACM Press, 1991, pp. 201–216.

[5] R. Amadio and D. Lugiez, "On the reachability problem in cryptographic protocols". Res. Rep. 3915, INRIA, 2000.

[6] R. Amadio and D. Lugiez, "On the reachability problem in cryptographic protocols", in *CONCUR'00, Lecture Notes in Computer Science.* Springer, 2000, vol. 1877.

[7] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman, "Magic sets and other strange ways to implement logic programs (extended abstract)", in *International Conference on Management of Data and Symposium on Principles of Database Systems.* ACM, 1986.

[8] D. A. Basin, "Lazy infinite-state analysis of security protocols", in *Secure Networking – CQRE (Secure) '99, International Exhibition and Congress*, R. Baumgart, Ed., *Lecture Notes in Computer Science*. Springer, 1999, vol. 1740, pp. 30–42.

[9] P. Bieber, "A logic of communication in hostile environment", in *Comput. Secur. Found. Worksh. (III)*, 1990, pp. 14–22.

[10] M. Burrows, M. Abadi, and R. Needham, "A logic of authentication". Tech. Rep. 39, Digital Equipment Corporation, Systems Research Centre, Febr. 1989.

[11] M. Burrows, M. Abadi, and R. Needham, "A logic of authentication", *ACM Trans. Comput. Syst.*, vol. 8, no. 1, pp. 18–36, 1990.

[12] A. Church, "An unsolvable problem of elementary number theory", *Am. J. Math.*, vol. 58, no. 2, pp. 345–643, 1936.

[13] D. Dolev and A. C. Yao, "On the security of public key protocols", *IEEE Trans. Inform. Theory*, vol. IT-29, no. 12, pp. 198–208, 1983.

[14] N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov, "Undecidability of bounded security protocols", in *Proc. Worksh. Form. Meth. Secur. Protoc. (FMSP)*, N. Heintze and E. Clarke, Eds., 1999.

[15] J. Y. Girard, *Proof and Types*. Cambridge University Press, 1990. (Translated and with appendices by Paul Taylor, Yves Lafont).

[16] L. Gong, "Cryptographic protocols for distributed systems". Ph.D. thesis, University of Cambridge, Cambridge, England, April 1990.

[17] L. Gong, R. Needham, and R. Yahalom, "Reasoning about belief in cryptographic protocols", in *IEEE Symposium on Research in Security and Privacy*. IEEE Computer Society Press, 1990, pp. 234–248.

[18] J. Hintikka, *Knowledge and Belief: An Introduction to the Logic of Two Notions*. Cornell University Press, 1962.

[19] D. Kindred and J. M. Wing, "Fast, automatic checking of security protocols", in *Second USENIX Workshop on Electronic Commerce*. USENIX, 1996, pp. 41–52.

[20] C. Meadows, "The NRL protocol analyzer: an overview", *J. Log. Program.*, 1995 (to appear).

[21] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. CRC Press, 1996.

[22] S. P. Miller, B. C. Neuman, J. I. Schiller, and J. H. Saltzer, "Kerberos authentication and authorization system", in *Project Athena Technical Plan*. MIT, 1987, ch. E.2.1.

[23] D. Monniaux, "Decision procedures for the analysis of cryptographic protocols by logics of belief", in *12th Computer Security Foundations Workshop*. IEEE, 1999.

[24] R. M. Needham and M. D. Schroeder, "Using encryption for authentication in large networks of computers", *Commun. ACM*, vol. 21, no. 12, 1978.

[25] D. Nessett, "A critique of the Burrows, Abadi and Needham logic", *ACM Oper. Syst. Rev.*, vol. 24, no. 2, pp. 35–38, 1990.

[26] L. C. Paulson, "Proving properities of security protocols by induction", in *10th Computer Security Foundations Workshop*. IEEE Computer Society Press, 1997, pp. 70–83.

[27] D. Prawitz, *Natural Deduction: a Proof-Theoretical Study*. Almqvist & Wiksell, Stockholm, 1965.

[28] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public key cryptosystems", in *SIMMONS: Secure Communications and Asymmetric Cryptosystems*, 1982.

[29] M. Rusinowitch and M. Turuani, "Protocol insecurity with finite number of sessions is NP-complete", in *14th IEEE Comput. Secur. Found. Worksh.*, Cape Breton, Nova Scotia, Canada, June 2001.

[30] B. Schneier, *Applied Cryptography*. 2 ed. Wiley, 1996.

[31] J. Schumann, "Automatic verification of cryptographic protocols with SETHEO", in *Proceedings of the 14th International Conference on Automated Deduction*, W. McCune, Ed., *Lecture Notes in Artificial Intelligence*. Berlin: Springer, 1997, vol. 1249, pp. 87–100.

[32] D. Song, "Athena: a new efficient automatic checker for security protocol analysis", in *12th Computer Security Foundations Workshop*. IEEE, 1999.

[33] P. Syverson, "Adding time to a logic of authentication", in *1st ACM Conf. Comput. Commun. Secur.*, 1993, pp. 97–101.

[34] P. Syverson and P. C. van Oorschot, "On unifying some cryptographic protocol logics", in *1994 IEEE Comput. Soc. Symp. Res. Secur. Priv.*, May 1994, pp. 14–28.

[35] P. Syverson, "Towards a strand semantics for authentication logics", in *Electronic Notes in Theoretical Computer Science*. 1999, vol. 20.

[36] P. Syverson and I. Cervesato, "The logic of authentication protocols", in *FOSAD'00*, R. Focardi and R. Gorrieri, Eds. Springer, 2001, vol. 2171.

[37] A. Tarski, "What is elementary geometry?", in *The Axiomatic Method, with Special Reference to Geometry and Physics*, P. Suppes and A. Tarski, Eds., *Studies in Logic and the Foundations of Mathematics*. North-Holand, 1959, pp. 16–29.

David Monniaux is a researcher at the Department of Computer Science at Ecole Normale Supérieure in Paris, France. His research interests include formal methods, program semantics and abstract interpretation, and the static analysis of cryptographic protocols, critical software, and probabilistic programs. He received his Ph.D. degree in computer science from Université Paris-Dauphine in 2002 and is an alumnus of Ecole Normale Supérieure de Lyon.
e-mail: David.Monniaux@ens.fr
École Normale Supérieure
Département d'Informatique
45, rue d'Ulm
75230 Paris Cedex 05, France

Paper

CardS4: modal theorem proving on Java smart cards

Rajeev Prabhakar Goré and Phuong Thê Nguyên

Abstract — We describe a successful implementation of a theorem prover for modal logic S4 that runs on a Java smart card with only 512 KBytes of RAM and 32 KBytes of EEPROM. Since proof search in S4 can lead to infinite branches, this is "proof of principle" that non-trivial modal deduction is feasible even on current Java cards. We hope to use this prover as the basis of an on-board security manager for restricting the flow of "secrets" between multiple applets residing on the same card, although much work needs to be done to design the appropriate modal logics of "permission" and "obligations". Such security concerns are the major impediments to the commercial deployment of multi-application smart cards.

Keywords — security of mobile code, modal deduction.

1. Introduction

Smart cards are credit-card sized pieces of plastic with an embedded silicon chip. Smart cards are either memory cards, which cannot be programmed, or microprocessor cards, which contain a small amount of RAM and disc (EEPROM) on the card itself. A card reader/writer is required to provide power to the card, to provide a clock signal, and to act as an interface between the card and the terminal (a PC, an ATM machine, a public telephone, or even a mobile telephone).

Java cards are smart cards that contain a (downsized) Java platform, installed by the manufacturer, thus allowing users to download Java applets and run them on the card. Java cards can therefore provide multiple applications such as electronic purse, credit card, passport, loyalty programmes, all residing on the same card.

While exchanging data securely already poses a number of problems (this prompts the need for cryptographic protocols, examined in a number of other papers in this issue), exchanging, even if only down-loading *programs* entails quite a number of new problems. It is all too easy to break every security policy by just down-loading one bad applet and letting it loose on the card. One example, not specific to cards, is the BrownOrifice applet [10], a Java applet that installs on any PC that down-loads and serves its entire file system to any outside attacker. Another is an attack allowing an outside intruder to register a bank transfer via the Quicken home-banking software [8].

The purpose of this paper is, first, to give a short survey of proposed techniques to enforce security in settings where applications can be loaded or down-loaded, and more specifically of Java card related techniques. Orthogonally, *logics*, and more specifically modal logics, have been used to specify security policies. We review the use of logics in this context. One challenge here is to be able to prove automatically formulae in sophisticated modal logics, efficiently, and – in the Java card context at least – under sparse memory resources. We shall demonstrate how this can be done for the logic **S4** – not yet the kind of logics we would like to deal with, but already one which is known to pose non-trivial problems. (Technically, this is because the transitivity of **S4** frames, as opposed to, say, **K** frames requires loop checks that are usually memory-consuming.) The Java cards used in this project were the GemXpresso RAD Protyping card, containing a 32-bit microprocessor with 512 bytes of RAM, 32 KBytes of Flash EEPROM and 8 KBytes of ROM. As of 2001, this is state of the art in Java card technology, and gives an indication of how little memory is available on Java cards.

The paper is set out as follows. We spend some time in Section 2 surveying method for ensuring security of mobile code, and Java card applets in particular. This includes discussions of several models or techniques, including the notion of non-interference, verification by typing, by static analysis of programs and by formal proofs in specific logics. We argue that being able to prove formulae of modal logics on-card is a promising security enforcement technique. The rest of the paper shows that, in principle, even complex modal logics with transitive frames can be handled on a card, by studying the prototypical transitive modal logic, **S4**. Section 3 describes the logic **S4** and the basics of modal theorem proving using tableaux. Section 4 explains the design of our prover `CardS4`, while Section 5 refines this by explaining the precise data structures that allow our prover to run in tiny memory spaces. Section 6 describes our implementation, and Section 7 presents test results. We conclude in Section 8.

2. Java cards and security of mobile code

2.1. Java cards

Current Java cards are preprogrammed to contain applets by the manufacturer for the card vendor, typically a bank (for credit and debit cards), or an airline (for frequent flyer cards). But if Java cards are to succeed then a card carrier must be able to down-load new applets onto an existing card "just in time", or even merge existing cards into one card. This would mean that multiple applets from different vendors would reside on the same card.

The single biggest problem with this scenario is that of security. How can we guarantee that a simple query to the drivers licence section of the card for identification purposes (say) will not steal money from the card's electronic purse? If new applets are to be down-loaded then how can the vendor of applet A ensure that a competing vendor's applet will not be down-loaded at a later stage and steal information from applet A? Alternatively, applets A and B may trust each other to some extent, and therefore share some information. But if applets B and C enjoy a similar trust relationship, how can A be sure that B will not tell C information which it has obtained from A [14, 34] ?

Many methodologies for guaranteeing such security have been investigated, but almost all of them involve a trusted "third" party. For example, the bank applet may be signed using a digital signature obtained from the government that certifies that the applet really did originate from the bank in question. The digital certificate is decoded by the card's on-board digital signature chip and the applet is allowed to access the card's electronic purse. But the need for a certification agency and a certification procedure makes this avenue cumbersome.

2.2. Enforcing security policies for mobile code

An alternative methodology that involves no third parties is for card owners to implement a personal security policy using some international standard "language for security". The electronic purse applet installed on the card may come with such a built in security policy which the user is prompted to tailor to his or her needs. Another applet which wishes to access the electronic purse must now pass a challenge determined by the level of security chosen by the card user.

As new applets are added to the card, they are slotted into this set up either explicitly by the card user, or by some implicit default method. The simplest method is to use some form of access control list as is done by the smart card for Windows system (http://www.microsoft.com/smartcard), which uses simple propositional logic in its access control lists. A more sophisticated approach is to use a hierarchy with the "public" applets at the bottom, the "private" applets at the top, and the others in between these two extremes in some partial order [14, 34]. This is similar to the Bell-La Padula model of military security [7], which is the basis of the access-right policy of operating systems like Unix. Each applet A is given an *accreditation* $acc(A)$ from some partially ordered set, while each object O on the card has an *access right* $right(O)$; the security policy is that A can only access O if $acc(A) \geq right(O)$; also A can only modify O, storing the contents of object O' into O, if $acc(A) \geq right(O)$ and $right(O') \leq right(O)$. This is Bell and La Padula's star condition; without it, A might unwittingly declassify O' by storing its contents into O, thereby allowing non-accreditated applets to access the contents of O' by subsequently reading that of O. These conditions can be enforced at run-time. The relationship between access control lists and object-level access rights is essentially a matter of whether access rights are stored in a centralized way or on a per-object basis. The precise relation between these and other so-called *trust management models*, such as capabilities, is analyzed in detail in [9].

The latter paper in particular addresses the difficult matters of handling *delegation*, whereby a subject (an applet, in our context) is allowed to act in the name of another for some designated objects, and *revocation*, whereby subjects are deprived of their accreditation. In the latter case, think of a vendor applet that will only provide a service to the card owner while she holds an annual subscription to that service.

Apart from revocation, trust management policies are still limited in the way they deal with dynamic change. In particular, how should the ordering $\geq$ be modified when a new applet is down-loaded? The Bell-La Padula model and its variants all assume a fixed ordering. But if down-loading an applet is allowed to modify the access rights ordering, possibly adding new accreditations or object access rights, a new policy is needed to prevent abuses; e.g., we should not allow the down-loading process to accept applets claiming to introduce a new accreditation greater than all pre-existing ones. Also, there should be some mechanism to enforce that the modified ordering still is an ordering.

Even then, solving these problems would not solve problems related to *transitive* information flows. Take the shared secrecy example above, where applet A trusts B, and B trusts C, but A does not trust C [14, 34]. For example, A might be a loyalty applet, and B might be a banking applet originating from a bank that has business deals with A's originator, so that if the card owner has accumulated enough loyalty points through A (think frequent flyer miles), then B will offer the card owner some added payment facilities. Now C might be the on-card part of an account management programme, which will need to access information from B, and will be trusted by B to do so. Then C can learn about the degree of loyalty of the card owner vis-a-vis A's originator by examining payment facilities offered by B: although A and C's originators never signed a deal allowing C to access loyality information from A, C can still get it through interaction with B.

2.3. Non-interference

Checking such properties can be done, at least partially, by checking *non-interference* properties [18]. At a basic level, non-interference for some computer system S (an applet, or a collection of applets) means that for every collection of objects O in the system, the value of objects with low access rights should never depend on the value of objects with high access rights. In other words, no observer should be able to tell anything about the values of objects with high access rights by just looking at values of objects with lower access rights. In particular, if an applet's accreditation level is a, its output should be independent of the value of any object with access right $> a$.

Non-interference is similar to the way secrecy and authentication are originally defined in Abadi and Gordon's spi-calculus [3]: a message M is *secret* in some protocol $P(M)$ if and only if no outside attacker can tell the difference between running $P(M)$ or running $P(M')$ for some other value M' of the secret. Formally, this is defined by saying that for every process I in the calculus, I parallel $P(M)$ and I parallel $P(M')$ should be may-testing equivalent. (Technically, this also requires the parallel compositions to be enclosed in suitably many (νn) constructs to represent channel, nonce and key generation.) This similarity with non-interference can be used to cast it as a non-interference problem, where the secret M would be assigned some high access right, and we require that non-interference holds assuming that the intruder has strictly lower accreditation. This is used in a typing system for secrecy [1]. A similar but slightly more complicated typing system also exists for authentication [17].

While non-interference seems a promising idea, *checking* non-interference is harder than it seems. Many type systems have been proposed in various restricted settings. One of the most sophisticated is [38], which considers non-interference in the presence of concurrency, where computations have observable durations, and in a probabilistic model.

Type systems for non-interference have to be crafted so that well-typed applets do obey non-interference. The resulting type systems are in general severely restricted as to which applets it will accept as well-typed.

To show what the difficulties are, first examine concurrency. Assume that applet A and applet B are both secure in the sense that none, when run alone, may terminate with some low object containing a value which depends on the initial contents of a high object. Then the parallel composition of A and B might be insecure. For example, consider three objects O_{hi}, O_{lo} and O'_{lo}, with respectively high, low and low access rights. For simplicity, assume that these objects only contain boolean values. Let A store the contents of O_{hi} inside O_{lo}, do nothing for a while, then erase all fields of O_{lo}; for short we write A as the program:

$$O_{lo} := O_{hi};\ \text{sleep};\ O_{lo} := 0;$$

A is secure, since the final value O_{lo}, zero, is independent of the initial value of O_{hi}. Let B do nothing for a while, test whether O_{lo} is true, and if so set O'_{lo} to true, otherwise to false. That is, B is:

$$\text{sleep; if } O_{lo} \text{ then } O'_{lo} :=\text{true else } O'_{lo} :=\text{false;}$$

Again, B is secure, since it never reads the value of any high object. Note that, if O_{lo} were replaced by O_{hi} in B, and even though B never copies its value to any other object, the resulting value of O'_{lo} would depend on that of O_{hi}, and B would not be secure; this shows that Bell and La Padula's conditions are in general not enough to ensure non-interference.

The important point in this example is that A and B in parallel are *not* secure: if A first does $O_{lo} := O_{hi}$, then B tests O_{lo}, thus in effect B is computing a value of O'_{lo} that depends on O_{hi}, violating non-interference. This may in fact happen with non-negligible probability, depending on the scheduler. The paper [38] examines more sophisticated interference patterns in which B cannot actually learn from the value of some high object because of timing considerations under a probabilistic model, assuming a probabilistically uniform scheduler. For example, if B sleeps long enough first in the example above, and A and B are started at the same time, then A parallel B will in fact still be secure with high probability.

2.4. Static program analysis

Checking properties of programs, whether security properties or others, can be done through typing, or through dataflow analysis, in general through any static program analysis technique.

One of the most well-known Java related dataflow analysis technique is Java's *bytecode verifier* [27]. Every downloaded Java class file, in particular every Java applet, is checked for format conformance first, then names are resolved, then every method in the class file is checked – this is bytecode verification proper. This latter phase checks that all operations are well-typed, that stacks do not overflow, plus a number of other sanity conditions, through a dataflow analysis.

While these checks are absolutely necessary for security (any type confusion error can indeed be exploited to create a security breach [30]), there are two issues that need to be addressed. First, the Java bytecode verifier consumes too many resources to be implemented on a Java card: in particular, the first Java cards did not include any bytecode verifier, and rested solely on cryptographic certificates and a trust relationship with applet issuers; as [8] demonstrates, this is not enough. Second, the bytecode verifier only addresses low-level safety issues (bounds checking, typing), and is far from ensuring any security-related property.

There are at least two different solutions to the first problem. One, inspired from Necula's *proof-carrying code* concept [32], is Rose's *lightweight bytecode verification* [36], used in Sun's small-footprint KVM Java virtual machine [39], designed for embedded applications. The idea is to split the bytecode verifier in an off-card part and an on-card part. The off-card verifier actually runs a dataflow analyzer similar to the standard bytecode verifier, except that on success it also outputs a certificate. The off-card verifier is run by the card issuer, who then appends the certificate to the applet. When the applet is down-loaded on the card, it comes with the certificate. The on-card verifier then only checks that the certificate is valid and is a certificate for the given applet. While cryptographic certificates are certainly the simplest form of certificates, the approach of [36] is more drastic: the certificates there are (a compressed form of) the typing information that a bytecode verifier needs to compute. It merely remains for the on-card verifier to check that the certificate is consistent with the semantics of all bytecode instructions present in each method. This takes less time, and more importantly less space than standard bytecode verification. This method applies to essentially any verification method that relies on proving some property of an applet in some formal deduction system (in the large; here typing is thought as a formal system, while Necula considers properties expressed in a variant of the logical framework LF [22] with a few extensions).

The other current way of incorporating bytecode verification into Java cards is Leroy's simplified bytecode verifier. This does not conform to Sun's specification of bytecode verifier as such, and in particular may reject applets that would be accept by the Sun's verifier. However this is repaired by an off-card component which rewrites any applet conforming to Sun's specification into one that will be accepted by the simplified verifier. The point is that the simplified verifier is actually able to run on a standard Java card, despite the severe restrictions on memory resources on cards. The basic intuitions behind this technique, as well as a lucid account of problems and solutions for bytecode verification, can be found in Leroy's paper [25].

The second problem with bytecode verification is that it only addresses low-level issues: typing, stack overflows, notably. It does not address any less trivial security issue such as the transitive flows mentioned earlier, for example. There is still little work on methods for checking more sophisticated security properties. Leroy and Rouaix [26] address the problem of verifying, by typing, that a down-loaded applet does not corrupt designated sensitive data on the system it is down-loaded onto. El Kadhi [5, 12] applies abstract interpretation methods to design a static analyzer that checks an applet for cryptographic confidentiality preservation properties: the goal is to ensure that designated sensitive data on a card are not leaked to a Dolev-Yao-style intruder (see [11]), even though this data may have to be sent out of the card (i.e., properly encrypted). This uses techniques from cryptographic protocol verification.

2.5. Logics

For checking more sophisticated security properties, it is implicit in the above discussion that we need a language to talk about the security properties of interest that can be understood both by card issuers and by on-card verifiers. This language should have a formal semantics. In other words, it should be a logic at large. Proof-carrying code already takes the viewpoint that properties should be specified in a logic, and that proofs should be sent along with the code to avoid costly reconstructions of proofs on the card side – this is a technological choice that may or may not be relevant, depending on the logic and available on-card resources. El Kadhi's work is another example: while the paper [12] does not mention any specific logic, El Kadhi's analyzer actually does deductions in a system of symbolic constraints that approximate the intruder's state of knowledge.

We can also use actual logics to express and check security properties. While this is the approach in Necula's original approach, taking a general logic such as LF might be overkill. In particular, LF provability is undecidable. However, multi-modal propositional logics provide interesting languages that are expressive enough to encode most properties of interest, while usually remaining decidable. Multi-modal propositional logics are now well-established in artificial intelligence research as bases for defeasible reasoning [37], logics of agents [35], and logics of authentication [4, 29]. Monniaux [31] shows that BAN and GNY logics are decidable, while Massacci [28] gives a tableaux calculus for the (undecidable) logic of access control of [2]. We refer the reader to [31] for more information on such logics. Multi-modal logics like Propositional Dynamic Logic [15] have also been used to model the changing states of a program. Finally, propositional bi-modal tense logics give a very simple and elegant model of the flow of time [21].

Checking that a down-loaded applet meets the security criteria is now reduced to proving, **on-board**, that an appropriate formula is a theorem of the logic used to code the criteria, since this is the only computer that the customer should trust. Let us stress that multi-modal logics are particularly well-suited to this task as most of them are decidable. Consequently, the ability to perform automated multi-modal deduction on Java smart cards may be of use in electronic commerce.

But surely multi-modal deduction is simply too difficult to perform on a smart card with extremely limited resources. After all, even classical propositional logic is NP-complete, and most multi-modal logics are actually PSPACE-complete!

In [19] automated deduction in bi-modal tense logics was shown to be feasible on a Java smart card. It is reasonably straightforward to extend this work to other multi-modal logics, and hence to logics of knowledge and belief, or to logics of authentication and security. But many of these logics (e.g. PDL) contain operators which are inherently transitive, and transitivity can lead to infinite loops. (This will be illustrated in later sections on **S4**.) In the sequel we show that transitivity is not insurmountable by implementing a prover for the transitive modal logic **S4**. This also shows that, although proof-carrying code-style techniques could be used here as well, they are probably unnecessary.

This work is naturally still far from an on-card prover for a logic of authentication or security, in the style of [4, 28]. This work should therefore be thought of as "proof of principle" that a logic-based security policy could be implemented on current Java cards. As the resources and speed of Java cards skyrocket, the task will only become simpler.

$w \models \top$		for every $w \in W$	$w \models \bot$		for no $w \in W$
$w \models p$	iff	$w \in V(p)$	$w \models \neg\varphi$	iff	$w \not\models \varphi$
$w \models \varphi \wedge \psi$	iff	$w \models \varphi$ and $w \models \psi$	$w \models \varphi \vee \psi$	iff	$w \models \varphi$ or $w \models \psi$
$w \models \varphi \to \psi$	iff	$w \not\models \varphi$ or $w \models \psi$			
$w \models \Diamond\varphi$	iff	$(\exists v \in R(w))(v \models \varphi)$	$w \models \Box\varphi$	iff	$(\forall v \in R(w))(v \models \varphi)$

Fig. 1. Kripke semantics for **S4**.

3. Syntax, semantics and tableaux for modal logic **S4**

3.1. Syntax and semantics for S4

Given a denumerably infinite set of atomic formulae $\texttt{PRP} = \{p_0, p_1, p_2, \cdots\}$, a formulae φ of modal logic is defined using the following BNF grammar:

$$\begin{array}{lcl} p & ::= & p_0 \mid p_1 \mid p_2 \mid \cdots \\ \varphi & ::= & \top \mid \bot \mid p \mid \neg\varphi_1 \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \to \varphi_2 \\ & & \mid \Diamond\varphi \mid \Box\varphi \end{array}$$

Propositional symbols in PRP denote elementary properties, e.g., "file `accounts` has access right `privileged`", or "user `Joe` has accreditation `standard`", or "`standard` $\geq$ `privileged`". They might be true or false; for example, we may imagine that the first two properties above are true, while the last one is false.

The other connectives are $\top$ (true), $\bot$ (false), $\neg$ (negation), $\wedge$ (and, conjunction), $\vee$ (or, disjunction), $\to$ (implication), and the modal connectives $\Diamond$ and $\Box$. The latter require some explanation. We may for example understand these connectives in the context of modeling agent knowledge (with one agent a) by letting $\Box A$ mean "a knows (is sure that) A", and $\Diamond A$ mean "a believes A". What the latter means is that a is not sure that A is false, so a accepts A as likely, although a cannot be sure of A.

The formulae of the logic **S4** can also be given another, temporal meaning, in which the truth-values of formulae evolve through time, and $\Box A$ means "from now on, A is always true", while $\Diamond A$ means "A will eventually become true at least once".

These meanings of **S4** formulae are special cases of its *Kripke semantics*. A Kripke frame is a pair $\langle W, R\rangle$ where W is a non-empty set (of worlds) and R is a binary relation over W. A Kripke model $\langle W, R, V\rangle$ is a Kripke frame $\langle W, R\rangle$ augmented with a valuation $V : \texttt{PRP} \mapsto 2^W$ mapping each atomic formula to the subset of W where they take the value "true". If $w \in V(p)$ we write $w \models p$ and extend this satisfaction relation to arbitrary formulae in the usual way [21] as shown in Fig. 1 where for any $w \in W$, $R(w) := \{v \in W \mid wRv\}$.

An **S4**-model is a Kripke model where R is both reflexive $(\forall w \in W)[wRw]$ and transitive $(\forall w_1, w_2, w_3 \in W)[w_1Rw_2 \,\&\, w_2Rw_3 \Rightarrow w_1Rw_3]$.

A formula φ is **S4**-*satisfiable* if and only if there exists some **S4**-model with some $w \in W$ such that $w \models \varphi$. A formula φ is **S4**-*valid* if $w \models \varphi$ for every $w \in W$ in every **S4**-model $\langle W, R, V\rangle$.

We illustrate this notion of model on a few **S4** formulae. First, $\Box\varphi \to \varphi$ is a valid formula. Temporally, this means that if from now on, φ is always true, then φ is true now. For agents, if a knows that φ holds, then φ indeed holds; that is, a does not make mistakes. Another interesting formula is $\Box\varphi \to \Box\Box\varphi$. Temporally, this means that if φ holds in every future from now, then in every future from now, in every future of this future, φ will again hold. In the world of agents, this is *positive introspection*: if a knows that φ holds, then a also knows that it knows that φ holds. A third important formula is $\Box(\varphi \to \psi) \to (\Box\varphi \to \Box\psi)$, which states that agents can perform deductions: if a knows that φ implies ψ, and a knows that φ holds, then a necessarily knows that ψ holds, too. Finally, we mention the subtle rule of necessitation: if φ is valid, then so is $\Box\varphi$. That is, if φ is always true, typically because there is a proof of φ, then a is sure that it holds. We let the logically-minded reader that the three formulae above and the necessitation rule all hold in any **S4**-model.

3.2. Proof search in S4 *using tableau calculi*

The problem of deciding whether or not a formula is **S4**-satisfiable is known to be PSPACE-complete [24]. The best known decision procedures use only $O(n^2.logn)$-space [23].

The most popular method for implementing theorem provers for **S4** is to use the tableau method [13, 16]. This uses the rules of Fig. 2, plus their duals for $\neg\Box$ and $\neg\Diamond$ obtained via the equivalences $\neg\Box\varphi = \Diamond\neg\varphi$ and $\neg\Diamond\varphi = \Box\neg\varphi$, and for negations of other connectives obtained via the equivalences $\neg(\varphi_1 \wedge \varphi_2) = \neg\varphi_1 \vee \neg\varphi_2$, $\neg(\varphi_1 \vee \varphi_2) = \neg\varphi_1 \wedge \neg\varphi_2$, $\neg\top = \bot$, $\neg\bot = \top$. The rules for implication are derived from $\varphi_1 \to \varphi_2 = \neg\varphi_1 \vee \varphi_2$.

An **S4**-tableau for a finite set of formulae Z is a binary tree of nodes where: the root node contains Z and the children are obtained by an application of some tableau rules for **S4** to the parent node. The rules are applied systematically so that the ($\Diamond$**S4**) rule is applied only to a *saturated* node: a node to which all other rules have already been applied. A branch of such an **S4**-tableau is closed if its leaf node contains both φ and $\neg\varphi$ for some formula φ, or if it contains $\perp$; otherwise the branch is open. The whole **S4**-tableau is closed if every branch in it is closed, otherwise it is open. A formula φ is *proved* when there is a closed tableau for the finite set $Z = \{\neg\varphi\}$.

$$\frac{X, \varphi_1 \wedge \varphi_2}{X, \varphi_1, \varphi_2}\,(\wedge) \qquad \frac{X, \varphi_1 \vee \varphi_2}{X, \varphi_1 \quad X, \varphi_2}\,(\vee)$$

$$\frac{X, \Box\varphi}{X, \Box\varphi, \varphi}\,(\Box\mathbf{S4}) \qquad \frac{X, \Box Y, \Diamond\varphi}{\Box Y, \varphi}\,(\Diamond\mathbf{S4})$$

Fig. 2. Tableau rules for **S4**.

As a first example, we may prove the formula

$$(p \wedge q) \rightarrow (q \wedge (q \rightarrow p))$$

as follows. First, negate this formula to get:

$$p \wedge q \wedge (\neg q \vee (q \wedge \neg p)).$$

Formally, we have simplified the negation by pushing negations inside formulae, using transformation rules $\neg\Box A \rightarrow \Diamond\neg A$, $\neg\Diamond A \rightarrow \Box\neg A$, $\neg(A \rightarrow B) \rightarrow A \wedge \neg B$, $\neg(A \wedge B) \rightarrow \neg A \vee \neg B$, $\neg(A \vee B) \rightarrow \neg A \wedge \neg B$, $\neg\top \rightarrow \perp$, $\neg\perp \rightarrow \top$, and removing double negations $\neg\neg A \rightarrow A$. This process ends in a formula where negation is only applied to atomic formulae, the so-called *negation normal form* (NNF).

Then, we may apply rule ($\wedge$) twice to produce the tableau node $p, q, \neg q \vee (q \wedge \neg p)$. The only rule that applies now is ($\vee$), yielding two nodes $p, q, \neg q$ and $p, q, q \wedge \neg p$. The first one is closed. The only rule that applies to the second is ($\wedge$), yielding $p, q, q, \neg p$, which is closed. Each branch is closed (contradictory): this terminates the proof. To sum up, this proof is written:

$$\cfrac{\cfrac{p \wedge q \wedge (\neg q \vee (q \wedge \neg p))}{p, q, \neg q \vee (q \wedge \neg p)}\,(\wedge)}{p, q, \neg q \qquad \cfrac{p, q, q \wedge \neg p}{p, q, q, \neg p}\,(\wedge)}\,(\vee)$$

One example we shall use in the sequel is:

$$\Box\Diamond p \wedge \Diamond\Box q \rightarrow \Diamond(q \wedge p). \tag{1}$$

Its temporal meaning is "if p is always such that it will become true later on, and if q eventually becomes true then remains true forever, then p and q will eventually become true simultaneously". Its meaning based on agent knowledge is "if I know that I believe p, and if I believe that I know q, then I believe p and q". This formula is **S4**-valid, as can be shown by looking at its Kripke semantics. Alternatively, we prove it as follows. A tableau proof starts with the NNF of its negation:

$$\Box\Diamond p \wedge \Diamond\Box q \wedge \Box(\neg q \vee \neg p). \tag{2}$$

A closed tableau is then:

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\Box\Diamond p \wedge \Diamond\Box q \wedge \Box(\neg q \vee \neg p)}{\Box\Diamond p, \Diamond\Box q, \Box(\neg q \vee \neg p)}\,(\wedge)}{\Box\Diamond p, \Diamond p^{(a)}, \Diamond\Box q^{(b)}, \Box(\neg q \vee \neg p), \neg q \vee \neg p}\,(\Box\mathbf{S4})}{\Box\Diamond p, \Box q, \Box(\neg q \vee \neg p)}\,(\Diamond\mathbf{S4})}{\Box\Diamond p, \Diamond p^{(a)}, \Box q, q, \Box(\neg q \vee \neg p), \neg q \vee \neg p}\,(\Box\mathbf{S4})}{\Box\Diamond p, p, \Box q, \Box(\neg q \vee \neg p)}\,(\Diamond\mathbf{S4})}{\Box\Diamond p, \Diamond p^{(a)}, p, \Box q, q, \Box(\neg q \vee \neg p), \neg q \vee \neg p}\,(\Box\mathbf{S4})}{\begin{array}{c}\Box\Diamond p, \Diamond p^{(a)}, p,\\ \Box q, q, \Box(\neg q \vee \neg p),\\ \neg q\end{array} \qquad \begin{array}{c}\Box\Diamond p, \Diamond p^{(a)}, p,\\ \Box q, q, \Box(\neg q \vee \neg p),\\ \neg p\end{array}}\,(\vee) \tag{3}$$

The first (topmost) use of rule ($\Box$**S4**) generates a node where there are two $\Diamond$-formulae, written here with superscripts (a) and (b). Then we may use ($\Diamond$**S4**) in two ways, using (a) or (b). The proof above uses (b); in fact there is no proof where we would use (a) instead at this point. We retrieve (a) below in the same proof: this is where it is used with ($\Diamond$**S4**). Although (a) is again regenerated below, we do not use ($\Diamond$**S4**) again. The final rule is ($\vee$), which closes the whole tableau. This proves (1).

Theorem 1 (Soundness and Completeness). The finite set $\{\neg\varphi\}$ has a closed **S4**-tableau iff the formula φ is **S4**-valid [13, 16].

As a special case, φ is **S4**-valid if and only if the finite set $\{\psi\}$, where ψ is the negation normal form of $\neg\varphi$, has a closed **S4**-tableau using only the rules of Fig. 2. Indeed a closed tableau for a negation normal form can only use the rules of Fig. 2. We shall restrict to negation normal forms in the rest of the paper.

The completeness proof gives a systematic method for proof-search, which consists of repeatedly applying all the invertible rules ($\wedge$), ($\vee$) and ($\Box$**S4**) until no more applications of these rules are possible. In the case of ($\Box$**S4**), this has to be made more precise: given $\Box\varphi$ in the current node, we add φ to it, and mark $\Box\varphi$ so as to prevent any reapplication of ($\Box$**S4**) to the same formula $\Box\varphi$. When none of these rules is applicable, we have reached a node that corresponds to a so-called saturated world in the underlying Kripke model under construction; see [16] for details. Some $\Diamond$-formula $\Diamond\varphi$ is then singled out for attention from this saturated node and a successor is created for it using the ($\Diamond$**S4**)-rule. Precisely, $\Diamond\varphi$ is replaced by φ, all $\Box$-formulae are unmarked (so as to reenable the application of ($\Box$**S4**)), and all other formulae are removed from the current node. The application of ($\Diamond$**S4**) usually requires backtracking: if no proof is found by singling out $\Diamond\varphi$ from the current node, some other $\Diamond$-formula has to be chosen instead, until one finds one that leads to a proof, or until all $\Diamond$-formulae have been tried.

Naive proof search for a closed **S4**-tableau for some finite set of formulae Z using this systematic method can lead to infinite loops viz. $Z = \{\Box\Diamond p, p\}$:

$$\begin{array}{cl} \Box\Diamond p, p & \\ \hline \Box\Diamond p, \Diamond p, p & (\Box\mathbf{S4}) \\ \hline \Box\Diamond p, p & (\Diamond\mathbf{S4}) \\ \vdots & \end{array}$$

Hence some form of *loop-checking* is necessary: if some node is obtained that has already been generated above in the same tableau, then proof search fails. In this example, this means that there is no closed tableau for Z.

Backtracking involves additional complications, in that loops do not always mean that there is no proof, rather that we have to make different choices to find a proof (if any). This can be seen from (3), where we can simulate the above loop by repeatedly applying rule $(\Diamond\mathbf{S4})$ on formula (a) (in particular, by using this rule rather than $(\vee)$ in the bottom deduction). This would loop; nonetheless (1) has a proof, namely (3).

Technically, it is also important that nodes are compared as sets, not just as lists; that is, duplicate formulae in nodes have to be removed. Otherwise, as the reader might want to check, the formula $\Box\Diamond\Box p \to \Diamond\Box\Diamond p$ leads to infinitely many nodes of the form $\Box\Diamond\Box p, \Box\Diamond\Box(\neg p), \Box p, \dots, \Box p$, with an unbounded number of occurrences of $\Box p$.

It turns out that sets of nodes obtained higher up by the $(\Diamond\mathbf{S4})$ rule only need to be kept in the checklist. As we shall demonstrate, this will allow us to keep the space requirements for proof search to within polynomial bounds.

4. Algorithms

We now describe our algorithm and data structures in more detail.

4.1. Terms

Negation normal form. Input formulae are assumed to be in negated normal form (NNF). This requirement is not restrictive since every formulae can be converted into a logically equivalent NNF formula in linear time [6]. The advantage of using NNF is that the formulae in the parse tree of the NNF formula constitute all of the formulae that can appear in any node of the search tree.

Parse tree. A formula is parsed as a tree, where each node has at most two children. The nodes are characterized as `CONJ`, `DISJ`, `ALL`, `SOME` if they are of type $\varphi \wedge \psi, \varphi \vee \psi, \Box\varphi, \Diamond\varphi$ respectively. At the leaves there are literals: atomic formulae or their negations. Each node represents a subformula of the original NNF formula, with the root representing the whole NNF formula. With the tableau rules of Fig. 2, it can be shown that the subformulae that appear in the parse tree are all the formulae that can appear in the nodes of the search tree. Clearly, the number of nodes in the parse tree is less than or equal to the length of the formula (it is exactly the length of the formula less the number of negation symbols). The number of formulae in any node of the search tree is therefore less than the length of the original NNF formula.

The parse tree is indexed, i.e., each node of the parse tree receives an integer number, so that each parent node has a smaller index than its children. This simplifies the visit sequence of the parse tree, as can be seen below.

Search tree. In the sequel we refer to the **S4** tableau as the search tree.

$n_\Diamond$, $n_\Box$, $n_\vee$, $n_\wedge$. The number of subformulae of the appropriate type in the original NNF formula.

4.2. Storing nodes in the search tree

The parse tree provides access to the finite list of all formulae that can appear in the nodes of the search tree. Thus each node in the search tree can be represented as a bit string, whose bits indicate whether or not the corresponding formulae is present in the node. This bit string has length equal to the length of the original formula. Thus storing one node in the search tree requires n bits, where n is the length of the original formula.

In the case of formula (2), we may index each subformula by the following numbers

$$\underbrace{\underbrace{\Box\underbrace{\Diamond\underbrace{p}_{7}}_{4}}_{1} \wedge \underbrace{\Diamond\underbrace{\Box\underbrace{q}_{8}}_{5}}_{2} \wedge \underbrace{\Box\underbrace{(\underbrace{\neg q}_{9} \vee \underbrace{\neg p}_{10})}_{6}}_{3}}_{0} \tag{4}$$

The leftmost final node of the proof (3) is then the set $\{1,4,7,5,8,3,9\}$, represented as the bit string 01110111010 (bit 0 being the rightmost), while the two conclusions of rule $(\Diamond\mathbf{S4})$ used in the proof are $\{1,5,3\}$ (00000101010) and $\{1,7,5,3\}$ (00010101010).

4.3. Loop checking

As shown in Section 3.2, a **S4**-tableau can contain an infinite branch. This problem can be solved by noticing that only a finite number of different nodes can appear in a search tree, and by avoiding examining any node twice. Thus if a node has ever been encountered before, it can be safely ignored. In this case, we backtrack to the last application of the $(\Diamond\mathbf{S4})$ rule, and choose a different $\Diamond$ formula there. If all $\Diamond$ formulae have been tried, we backtrack higher up to the previous application of the $(\Diamond\mathbf{S4})$-rule, and so on until all avenues have been explored,

$$\begin{array}{rcll}
\texttt{prove}(pos, neg, \bot :: \ell, boxes, dias, checkList) & = & true & \\
\texttt{prove}(pos, neg, \top :: \ell, boxes, dias, checkList) & = & \texttt{prove}(pos, neg, \ell, boxes, dias, checkList) & \\
\texttt{prove}(pos, neg, p :: \ell, boxes, dias, checkList) & = & p \in neg \vee \texttt{prove}(pos \cup \{p\}, neg, \ell, boxes, dias, checkList) & (p \in \texttt{PRP}) \\
\texttt{prove}(pos, neg, \neg p :: \ell, boxes, dias, checkList) & = & p \in pos \vee \texttt{prove}(pos, neg \cup \{p\}, \ell, boxes, dias, checkList) & (p \in \texttt{PRP}) \\
\texttt{prove}(pos, neg, (\varphi \wedge \psi) :: \ell, boxes, dias, checkList) & = & \texttt{prove}(pos, neg, \varphi :: \psi :: \ell, boxes, dias, checkList) & \\
\texttt{prove}(pos, neg, (\varphi \vee \psi) :: \ell, boxes, dias, checkList) & = & \texttt{prove}(pos, neg, \varphi :: \ell, boxes, dias, checkList) & \\
 & \wedge & \texttt{prove}(pos, neg, \psi :: \ell, boxes, dias, checkList) & \\
\texttt{prove}(pos, neg, \Diamond\varphi :: \ell, boxes, dias, checkList) & = & \texttt{prove}(pos, neg, \ell, boxes, dias \cup \{\varphi\}, checkList) & \\
\texttt{prove}(pos, neg, \Box\varphi :: \ell, boxes, dias, checkList) & = & \texttt{prove}(pos, neg, \varphi :: \ell, boxes \cup \{\varphi\}, dias, checkList) & \\
\texttt{prove}(pos, neg, [], boxes, dias, checkList) & = & \exists \varphi \in dias \cdot (boxes, \varphi) \notin checkList & \\
 & & \quad \wedge \texttt{prove}(\emptyset, \emptyset, \varphi :: boxes, boxes, \emptyset, checkList \cup \{(boxes, \varphi)\}) &
\end{array}$$

Fig. 3. A straightforward proof search algorithm.

or a closed **S4**-tableau is found. This, however, is not a practical method, since it would require exponential space to store all the possible nodes of the search tree.

A better method is to check for repetitions of the nodes obtained by the application of the $(\Diamond\mathbf{S4})$ rule only, since these contain the "core" of the new worlds built by this rule. Thus the latter method looks for repetitions of the initial configuration of each newly generated world. This also guarantees the solution for the infinite branch problem, yet requires polynomial space. The price is that identical nodes on different branches now have to be treated separately.

4.4. A straightforward non-deterministic algorithm

A naive implementation of **S4** tableaux would be by the following recursive procedure `prove`. We specify it concretely enough that we can use it as actual code in any functional programming language. The `prove` function takes six arguments $(pos, neg, \ell, boxes, dias, checkList)$, where *pos* and *neg* are sets of propositional variables (being variables p occurring as p, resp. $\neg p$ on the current node), ℓ is a list of formulae (the part of the current node containing formulae that we have to deal with), *boxes* is a set of formulae φ such that $\Box\varphi$ occurs in the current node, *dias* is a set of formulae φ such that $\Diamond\varphi$ occurs in the current node – so that the current node is exactly the set of formulae in ℓ, plus all atoms from *pos*, all negations of atoms from *neg*, all formulae of *boxes* with a $\Box$ added in front, and all formulae of *dias* with a $\Diamond$ added in front. Finally, *checkList* is a set of pairs $(boxes, \varphi)$ representing conclusions of the $(\Diamond\mathbf{S4})$ rule that we have encountered higher up in the current tableau and which should not repeat (loop-checking).

Then our first algorithm may be written as in Fig. 3, which may be implemented right away in languages like ML [33]. The notation $\varphi :: \ell$ denotes the list ℓ with φ added in front. This is mainly used for pattern-matching purposes, and in that case it checks that the corresponding argument is a non-empty list, gets the first element in φ and the rest of the list in ℓ. We write [] the empty list, and [*boxes*] any list whose elements are those of the set *boxes*.

All the clauses in Fig. 3 except the last one dispatch formulae in the to-do list ℓ depending on their topmost symbol. Eventually, `prove` will be called with $\ell = []$, leading to the last clause, which tries to apply rule $(\Diamond\mathbf{S4})$ in a way leading to a proof. The condition $(boxes, \varphi) \notin checkList$ implements loop-checking, together with the fact that we add $(boxes, \varphi)$ to *checkList* in the last call to `prove`. (It would have been more natural, by the way, to call `prove` on arguments $(\emptyset, \emptyset, \varphi :: [\Box\psi | \psi \in boxes], \emptyset, \emptyset, ...)$; Figure 3 instead directly precompiles the obvious applications of rule $(\Box\mathbf{S4})$).

Finally, to prove φ, call $\texttt{prove}(\emptyset, \emptyset, Z, \emptyset, \emptyset, \emptyset)$, where Z is the one-element list containing the NNF of φ.

4.5. A non-deterministic algorithm

Figure 3 is quite detailed, and we shall rather use more informal notation in the sequel, so as to concentrate on the essentials. We shall also adopt a more imperative style of writing, writing *currentWorld* for the current node and assigning to it, and using the phrase "non-deterministically do" to replace existential quantifications (as in the last clause of the figure).

Since card memory is scarce, we also need to control how much space is used, and in particular it is dangerous to use a recursive style. Instead, we manage the recursion stack by hand. This saves space for many useless local variables and for return addresses: we don't need to memorize either kind of object. So our stack *stack* will only contain pairs of a checklist and a formula to backtrack to (these will always be left arguments to $\vee$). As an optimization, the sequential nature of the algorithm and the fact that check lists always grow as a tableau expands ensure that we need not store entire checklists: a single array *checkList* suffices, and the stack only memorizes which prefix of *checkList* is actually relevant: this prefix is coded as an integer *checkListSize* counting the number of initial objects in *checkList* that consistute the actual check list.

The following algorithm returns true if a given formula φ_0 of length n is **S4**-satisfiable, and false otherwise. It requires n^4 space.

```
$stack$ ::= empty
$checkList$ ::= empty
$checkListSize$ ::= 0
$currentWorld$ ::= $\{\varphi_0\}$
```

```
do
    while there are $\wedge$, $\vee$, $\Box$ rules that can be applied do
        apply these rules to currentWorld
        if the rule applied is an $\vee$ rule then
            push (checkListSize, left child of the rule)
                        onto stack
            currentWorld ::= the right child of rule
        end if
    end while
    if currentWorld is closed then
        if the stack is empty then
            return false
        else
            (checkListSize, currentWorld) ::=
                        pop from stack
            resize checkList to checkListSize
            continue
        end if
    end if
    if currentWorld appears in checkList[0..checkListSize-1]
        then return true
    end if
    if no ($\Diamond$S4)-rules can be applied then
        return true
    end if
    non-deterministically pick a formula $\Diamond\varphi$
            from currentWorld
    apply ($\Diamond$S4)-rule using $\Diamond\varphi$ to currentWorld
            to get newWorld
    checkList[checkListSize] ::= currentWorld
    checkListSize::= checkListSize+1
    currentWorld ::= newWorld
while stack is not empty
```

Space complexity. It can be seen that *stack* grows by 1 only when the ($\vee$) rule is applied. Thus for one world, we may need to store the maximum of $n_\vee$ possible configurations. Also the transitional rule ($\Diamond$**S4**) which moves from one world to another preserves all the $\Box$-formulae, thus the set of all $\Box$-formulae (the core) in consecutive worlds in a branch of the search tree do not decrease. Therefore there are at most $n_\Box$ different cores in the same branch of the search tree. Also the worlds that have the same core will form a chain in the branch. We need to find the maximum number of worlds in a chain that have the same core. Since each new world is formed by taking all the $\Box$ formulae from the previous world together with one of the $\Diamond$ formulae, the maximum number of worlds in the chain that have the same core is $n_\Diamond$. Altogether, the maximum number of configurations that we need to store in the stack is $n_\vee \times n_\Box \times n_\Diamond \leq n^3$ (a better approximation is $n^3/3$).
It can be seen that *checkList* contains only the node which is obtained by applying a ($\Diamond$**S4**) rule. It is also resized so that all the nodes it contains are the initial configurations of the worlds in the current search branch. Thus *checkList* is always smaller than or equal to *stack*. Overall, the maximum number of configurations we might need to store in *stack* and *checkList* is of order n^3. Given that a configuration needs n bits, the space complexity of this algorithm is n^4 [24].

4.6. An improved algorithm

First, the non-deterministic choice of the final part of the algorithm must be eliminated: some form of enumeration of $\Diamond$ formulae has to be implemented. For each $\Diamond$ formula $\Diamond\varphi$ in *currentWorld*, we have to generate the initial configuration X_φ of a new world by just keeping φ and all $\Box$ formulae from *currentWorld*, and proceeding to build a closed sub-tableau for X_φ. If some such attempt succeeds, then *currentWorld* is refuted. To enumerate the formulae $\Diamond\varphi$, we store the world configuration *currentWorld* before we apply the ($\Diamond$**S4**) rule and keep the index *lastK_index* of the last ($\Diamond$**S4**) rule applied to that configuration.
Second, it is safe to always use the ($\vee$) rule before any instance of ($\Diamond$**S4**), as done in Section 4.5, but also to only apply ($\vee$) once all rules ($\wedge$) and ($\Box$) have been applied. While this is not always optimal, it is generally a good heuristic: for example, the proof (3) never uses ($\vee$) before ($\Diamond$**S4**) except at the last step. Indeed, with respect to Fig. 3, the ($\vee$) rule involves a form of "universal" backtracking while ($\Diamond$**S4**) involves a form of "existential" backtracking, and it is usually better to postpone backtracking rules as much as possible.
Third, while backtracking usually involves memorizing one branch of the computation on the stack while we explore the other branch, there is no need to memorize anything when backtracking is caused by instances of ($\vee$), provided we know which subformulae in the current node are first or second arguments to an $\vee$ in the whole formula to prove, and provided we exploit certain properties of our indexing scheme for subformulae. Let us call a formula of type R if it occurs as a second argument to $\vee$, and of type L if it occurs as a first argument to $\vee$.
This is best explained on an example. Consider the node:

$$\underbrace{\Box\Diamond p}_{1}, \underbrace{\Diamond p^{(a)}}_{4}, \underbrace{\Box q}_{5}, \underbrace{q}_{8}, \underbrace{\Box(\neg q \vee \neg p)}_{3}, \underbrace{\neg q \vee \neg p}_{6} \qquad (5)$$

which we have already encountered in the proof (3). Note that $\neg q$ (subformula 9) is the only type L subformula, while $\neg p$ (subformula 10) is the only type R subformula, and we may go from one to the other by incrementing, resp. decrementing the index. This is because subformulae were actually indexed in a breadth-first manner.
To look for a closed tableau from this node, set *lastK_index* to 0, and generate the first of the conclusions of the ($\vee$) rule:

$$\underbrace{\Box\Diamond p}_{1}, \underbrace{\Diamond p^{(a)}}_{4}, \underbrace{\Box q}_{5}, \underbrace{q}_{8}, \underbrace{\Box(\neg q \vee \neg p)}_{3}, \underbrace{\neg q}_{9} \qquad (6)$$

Note that we do *not* stack the other conclusion of the ($\vee$) rule. Once we reach a closed node for this node, we know that we can obtain a corresponding node (not necessarily closed) of the subtableau below the other conclusion of the ($\vee$) rule, by replacing the first type L subformula by the corresponding type R subformula (the unique other argument to the same $\vee$ operator), and all preceding type R subformulae by the corresponding type L subformula. If there is no type L subformula remaining, then all conclusions of the $(\vee)$ rule have been dealt with. Here the only type L subformula is 9, so we produce:

$$\underbrace{\Box\Diamond p}_{1},\underbrace{\Diamond p^{(a)}}_{4},\underbrace{\Box q}_{5},\underbrace{q}_{8},\underbrace{\Box(\neg q\vee\neg p)}_{3},\underbrace{\neg p}_{10} \qquad (7)$$

Now no other rule than $(\Diamond\mathbf{S4})$ applies: *lastK_index* is incremented to 4, the position of formula (a). We get:

$$\underbrace{\Box\Diamond p}_{1},\underbrace{p}_{7},\underbrace{\Box q}_{5},\underbrace{\Box(\neg q\vee\neg p)}_{3} \qquad (8)$$

which is the initial configuration of the new world. In particular, we memorize $\{1,7,5,3\}$ into *checkList*. We must also reset *lastK_index* to 0, lest we lose required opportunities for applying $(\Diamond\mathbf{S4})$ later on.
Apply all $\wedge$ and $\Box\mathbf{S4}$ rules, getting:

$$\underbrace{\Box\Diamond p}_{1},\underbrace{\Diamond p^{(a)}}_{4},\underbrace{p}_{7},\underbrace{\Box q}_{5},\underbrace{q}_{8},\underbrace{\Box(\neg q\vee\neg p)}_{3},\underbrace{\neg q\vee\neg p}_{6} \qquad (9)$$

Again apply $(\vee)$, leading to the closed node:

$$\underbrace{\Box\Diamond p}_{1},\underbrace{\Diamond p^{(a)}}_{4},\underbrace{p}_{7},\underbrace{\Box q}_{5},\underbrace{q}_{8},\underbrace{\Box(\neg q\vee\neg p)}_{3},\underbrace{\neg q}_{9} \qquad (10)$$

This is closed, so change the type L formula $\underbrace{\neg q}_{9}$ into the corresponding type R formula $\underbrace{\neg p}_{10}$. This is now closed again, and no type L formula remains. The proof is finished.
We let the interested reader do the full example (2), noticing that loop-checking is now required.
Notice that our scheme for avoiding storing information for backtracking on $\vee$ subformulae requires us to index each subformula of the original formula with a distinct number, even though some subformulae may be equal. This indexing of the nodes of the parse tree forbids common sub-expressions and therefore introduces some redundancy. There may be duplications of the same subformula, but they are named with different indices, so they must be examined independently.
In the following algorithm, the stack stores only the configurations that have been fully expanded with non-$(\Diamond\mathbf{S4})$ rules. Also, "saturate *currentWorld* with non-$(\Diamond\mathbf{S4})$ rules" means applying $(\wedge)$, $(\Box\mathbf{S4})$ and the left part of $(\vee)$:

$$\frac{X,\varphi_1\vee\varphi_2}{X,\varphi_1}$$

to *currentWorld* while this changes *currentWorld*. Recall that the right part of the $(\vee)$ rule will be obtained by looking at remaining type L formulae in the initial configuration of the current world, what we call "another sibling of *currentWorld*" below.

```
checkList ::= empty
stack ::= empty
currentWorld ::= world consisting of the original formula
do
    1. Saturate currentWorld with non-($\Diamond$S4) rules
    2. while currentWorld is closed do
        if there is another sibling of currentWorld then
            take currentWorld to be that sibling
            reset lastK_index to indicate no $\Diamond$S4 rules
                have been applied yet
        else if the stack is empty then
            return false
        else
            (lastK_index,currentWorld) ::=
                pop from stack
            pop from checkList
        end if
      end while
    3. do
        lastK_index ::= the next $\Diamond$-formula
            from currentWorld
        if no more ($\Diamond$S4) rules can be applied then
            if stack is empty then
                return true
            else
                (lastK_index,currentWorld) ::=
                    pop from stack
                pop from checkList
            end if
        end if
        apply the $\Diamond$S4 rule to currentWorld
            to get newWorld
      while newWorld appears in checkList
    4. put (lastK_index,currentWorld) into stack
    put newWorld into checkList
    currentWorld ::= newWorld
while stack is not empty
```

Note that now *checkList* and *stack* are of the same size. The *checkList* is actually the core of the world configuration stored at the corresponding location in *stack*. Thus it can be obtained from *stack* by generating the core of each world in *stack*. This gives a more efficient use of space. It can be seen that for one node in the search tree, we need only one location in *stack*. Thus the space requirement is reduced by a factor of $n_\vee$: the maximum number of configurations stored in *stack* at one time is $n_\Box\times n_\Diamond\le n^2$, and the space complexity of this algorithm is n^3 (a better approximation is $n_\Box\times n_\Diamond\le n^2/2$, and the space requirement is $n^3/2$).

5. Data structures

5.1. The parse tree

The parse tree is stored in two byte arrays, one (*childs*) of length $2n$ and the other (*nature*) of length n. The $2i$ and $2i+1$ entries in the first array indicate the children of the *i*th node in the parse tree (i.e. the indices of some other nodes in the parse tree, or the atom at that node), while the *i*th entry in the second array indicates if the *i*th node in the parse tree is a $\Box$, $\Diamond$, $\vee$, $\wedge$, $\neg$ or `PRP`. The $\vee$ and $\wedge$ nodes have two children. The $\Box$, $\Diamond$, $\neg$ and `PRP` nodes have one child, thus the second child for these node is redundant. This is not a severe problem, since the parse tree is fixed through the proving procedure. Note that in case of the $\neg$ and `PRP` nodes, the $2i$ entry of the *child* array contains the atom itself, not the index to another node in the parse tree.

5.2. The nodes in the search tree

As discussed above, each node in the search tree can be represented as a bit string of length *n*. To examine all the ($\Diamond$**S4**)-rules that can be applied to a node (i.e all the $\Diamond$ formulae in that node), we need to store the index to the last $\Diamond$ formula that has been examined, requiring one more byte.

5.3. The stack

Nodes are stored in a stack so that other branches in the search tree can be generated from the current branch, and so that a new node can be checked for duplication. This requires searching through all nodes in the stack, and also pushing and popping from the stack.
There are several possible implementations for the stack. The first two options store the stack as an array of bytes, and do not need extra memory for pointers. Since multidimensional arrays are not supported, access to elements of the stack will require some extra computations.
The upper bound of the size of the stack is known as seen above. Thus the stack can be allocated at the beginning of the procedure. We then need not worry about the growth of the stack. However, this is not a practical method, since the stack rarely grows to its theoretical upper bound size. Also, the limited amount of memory on the card will restrict the size of the input formulae. For example, with 512 bytes, the length of the formula will be less than 16, i.e. $16^3/2$ bits = = 512 bytes. Allocating more than the card's RAM is possible, however it involves swapping to and from the EEPROM and will slow down the proof procedure. Consequently, the card reader usually cannot wait for the card and will throw an exception.
Another way to implement the stack is to pre-allocate a small stack, and gradually increase its size by a large step when it becomes full. This ensures that the memory is used more effectively. However it still contains redundancy since it allocates more space than required each time it becomes full. Thus the longer formulae will result in larger redundancy. It also involves a lot of copying each time the stack grows.
The stack can also be implemented as a one way link list. Each node in the search tree is an element of the list. This requires extra memory for a pointer to the next element. However this extra memory becomes insignificant for long formulae. There is no redundancy. With this approach, the program has been tested for formulae of length up to 120.

6. Implementation

The program consists of two packages: `card` and `client`. The `client` package contains the classes for parsing the formula and converting it into NNF. Parsing is done by using Javacup (version 1.0j). We need a scanner (`scanner.java`) and a specification (`parser.cup`) for the formula, and Javacup automatically creates the parser. There is also a card proxy which manages the interactions with the card on behalf of the users. The class for testing is also in this package. Note that all of these operations are done off-board on a terminal (PC).
The `card` package contains an interface and two classes that are downloaded onto the card. These are classes `prover` and `State`, and the interface `proverInterface`. The interface `proverInterface` provides access to the services offered by the prover. These include loading the formula onto the card and proving. The interface also defines constants that are used by the prover, and are also used in the parsing and converting procedures. The class `prover` contains the codes for the proving procedure. The `prover` object that is loaded onto the card reserves enough space to hold the longest formula. When the formula is put onto the card, it is stored in the object. The user then must explicitly call the `prove` procedures. The `prove` method reads the formula from the object, performs simplifications and then starts looking for a model for the formula. (Note that there is a separation between loading the formula and proving. This is due to the fact that loading is rather complicated, and it is discussed in the next paragraph).
Despite the limitations of the card, the prover is able to work for a number of long formulae. Tests have been conducted for formulae of length up to 120. Passing the input to the card and storing input in the card then requires greater care because communication with the card is not simple. There is an upper-bound for the amount of data that can be transfered in one transmission (approaching 64K is not recommended). Long formulae therefore need to be broken into small pieces. Here, the input arrays are split into pieces of length 32 bytes and each piece is passed separately to the card, together with its length and position in the original array. Thus the maximum amount of data in one transmission is 34 bytes (one byte for the length and one for the position).
Since the inputs to the `prove` method are not ready in one pass, they need to be stored in the object `prover`, requiring the reservation of space for the longest input. Note that this also implies more time is required for copying the input from EEPROM to RAM in the `prove` procedure.

7. Results

This section shows the average time spent on the card in proving randomly generated formulae of various lengths

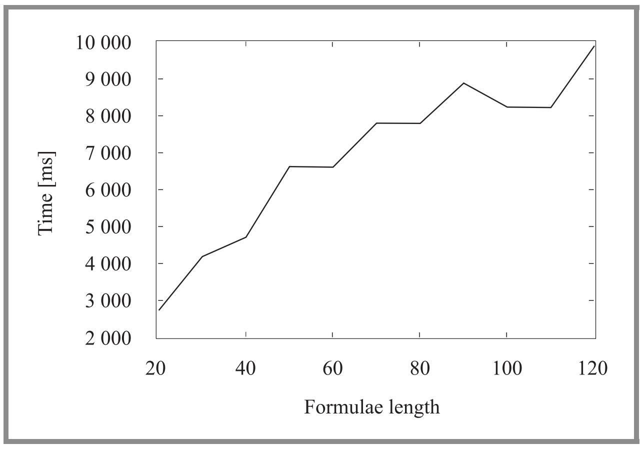


Fig. 4. Time vs formulae length.

(from 20 to 120). As can be seen, the time increases with the length of the formulae since longer formulae generally require more stack space, and require more arithmetic operations in the calculations (Fig. 4).

8. Conclusions and further work

We have shown that even modal logic **S4** can be handled on a Java card. Thus transitive modal logics are not necessarily beyond the scope of Java cards. We now need to invent or explore appropriate logics of permissions and obligations to allow us to capture basic security notions like "trust". This is the subject of further work.
Another method for loop checking is to keep track of certain formula using a history mechanism [20]. We intend to investigate whether such a history mechanism can be easily used in `CardS4`.

Acknowledgements

We are grateful do Didier Tollé of Gemplus Australia for donating the hardware necessary to carry out this project.
The authors acknowledge support from: Rajeev Prabhakar Goré – Queen Elizabeth II Fellowship from the Australian Research Council, Phuong Thê Nguyên – RSISE Summer Research Scholarship.

References

[1] M. Abadi, "Secrecy by typing in security protocols", *J. ACM*, vol. 46, no. 5, pp. 749–786, 1999.

[2] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin, "A calculus for access control in distributed systems", *ACM Trans. Program. Lang. Syst.*, vol. 15, no. 4, pp. 706–734, 1993.

[3] M. Abadi and A. D. Gordon, "A calculus for cryptographic protocols: the spi calculus", in *Fourth ACM Conference on Computer and Communications Security*. ACM Press, 1997.

[4] M. Burrows, M. Abadi, and R. Needham, "A logic of authentication", *ACM Trans. Comput. Syst.*, vol. 8, pp. 18–36, 1990.

[5] P. Boury and N. El Kadhi, "Static analysis of Java cryptographic applets", in *ECOOP'2001 Workshop on Formal Techniques for Java Programs*. Tech. Rep., FernUniversität Hagen, 2001. [Online]. Available: http://www.informatik.fernuni-hagen.de/import/pi5/ workshops/ecoop2001_papers.html.

[6] N. Bonnette and R. Goré, "A labelled sequent system for tense logic Kt", in *AI98: Proceedings of the Australian Joint Conference on Artificial Intelligence, LNAI*. Springer, 1998, vol. 1502, pp. 71–82.

[7] D. Bell and L. La Padula, "Secure computer systems: unified exposition and multics interpretation". Tech. Rep., MITRE Corp., July 1975, no. MTR-2997.

[8] K. Brunnstein, "Hostile activeX control demonstrated", *RISKS Forum*, vol. 18, no. 82, 1997.

[9] A. Chander, D. Dean, and J. Mitchell, "A state-transition model of trust management and access control", in *14th Computer Security Foundations Workshop*. IEEE Computer Society, 2001, pp. 27–43.

[10] CERT Coordination Center. CERT$^{(R)}$ Advisory CA-2000-15. "Netscape allows Java applets to read protected resources". [Online]. Available: http://www.cert.org/advisories/CA-2000-15.html.

[11] H. Comon and V. Shmatikov, "Is it possible to decide whether a cryptographic protocol is secure or not?", *J. Telecommun. Inform. Technol.*, no. 4, pp. 5–15, 2002.

[12] N. El Kadhi, "Automatic verification of confidentiality properties of cryptographic programs", in *Networking and Information Systems*, vol. 3, no. 6, Hermès, 2001. [Online]. Available: http://www.epita.fr:8000/ el-kad_n/Hermes.ps.

[13] M. Fitting, *Proof Methods for Modal and Intuitionistic Logics*, in *Synthese Library*, D. Reidel, Ed. Dordrecht, Holland, 1983, vol. 169.

[14] P. Girard, "Which security policy for multiapplication smart cards", in *Proc. USENIX Worksh. Smart Card Technol.*, Chicago, USA, 1999, pp. 21–28.

[15] R. I. Goldblatt, *Logics of Time and Computation*, *CSLI Lecture Notes*. Stanford: Center for the Study of Language and Information, 1987, no. 7.

[16] R. Goré, "Tableau methods for modal and temporal logics", in *Handbook of Tableau Methods*, M. D'Agostino, D. Gabbay, R. Hänle, and J. Posegga, Eds. Kluwer, 1999, ch. 6, pp. 197–396.

[17] A. Gordon and A. Jeffrey, "Authenticity by typing for security protocols", in *14th Computer Security Foundations Workshop*. IEEE Computer Society, 2001, pp. 145–159.

[18] J. A. Goguen and J. Meseguer, "Security policies and security models", in *IEEE Symp. Secur. Priv.*, 1982.

[19] R. Goré and L. D. Nguyen, "CardKt: automated multi-modal deduction on Java cards for multi-application security", in *Proc. Java Card Workshop, LNCS*. Springer, 2001 (to appear).

[20] A. Heuerding, "Automated deduction in some propositional modal logics". Institut für Angewandte Mathematik und Informatik, Universitäte Bern, Switzerland, 1999.

[21] G. E. Hughes and M. J. Cresswell, *A New Introduction to Modal Logic*. Routledge, 1996.

[22] R. Harper, F. Honsell, and G. Plotkin, "A framework for defining logics", *J. Assoc. Comput. Machin.*, vol. 40, no. 1, pp. 143–184, 1993.

[23] J. Hudelmaier, "Improved decision procedures for the modal logics K, T and S4".

[24] R. Ladner, "The computational complexity of probability in systems of modal propositional logic", *SIAM J. Comput.*, vol. 6, no. 3, pp. 467–480, 1977.

[25] X. Leroy, "Java bytecode verification: an overview", in *Proceedings of 13th International Conference on Computer-Aided Verification*, G. Berry, H. Comon, and A. Finkel, Eds., *Lecture Notes in Computer Science*. Springer, 2001, vol. 2102, pp. 265–285.

[26] X. Leroy and F. Rouaix, "Security properties of typed applets. Secure Internet programming – security issues for mobile and distributed objects", in *Lecture Notes in Computer Science*. Springer, 1999, vol. 603, pp. 147–182. [Online]. Available: http://pauillac.inria.fr/~xleroy/publi/sip-typed-applets.ps.gz.

[27] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*. The Java series. 2nd ed. Addison-Wesley, 1999.

[28] F. Massacci, "Tableaux methods for access control in distributed systems", in *Automated Reasoning with Analytic Tableaux and Related Methods*, D. Galmiche, Ed., *Lecture Notes in Artificial Intelligence*. Springer, 1997, vol. 1227, pp. 246–260.

[29] A. Mathuria, "Contributions to authentication logics and analysis of authentication protocols". Ph.D. thesis, School of Information Technology and Computer Science, University of Woolongong, Woolongong, Australia, 1997.

[30] G. McGraw and E. Felten, *Securing Java – Getting Down to Business with Mobile Code*. Wiley, 1999. [Online]. Available: http://www.securingjava.com/.

[31] D. Monniaux, "Analysis of cryptographic protocols using logics of belief: an overview", *J. Telecommun. Inform. Technol.*, no. 4, pp. 57–67, 2002.

[32] G. Necula, "Proof-carrying code", in *Conference Record of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 1997, pp. 106–119.

[33] L. C. Paulson, *ML for the Working Programmer*. 2nd ed. Cambridge University Press, 1996.

[34] P. Girard, J.-L. Lanet, V. Wiels, G. Zanon, P. Bieber, and J. Cazin, "Checking secure interactions of smart card applets". Tech. Rep., Gemplus R&D Centre, 2000. [Online]. Available: http://www.gemplus.com/smart/r_d/projects/pacap.htm.

[35] A. Rao and M. Georgeff, "A model-theoretic approach to the verification of situated reasoning systems", in *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI-93)*. Morgan-Kauffman, 1993, pp. 318–324.

[36] E. Rose and K. Rose, "Lightweight bytecode verification", in *OOPSLA Satel. Worksh. Form. Underpin. Java*, 1998.

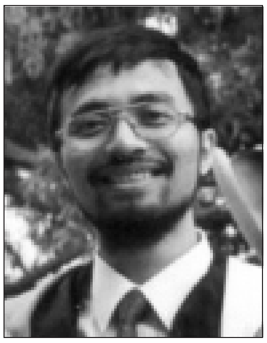
[37] G. F. Shvarts, "Autoepistemic modal logics", in *Theoretical Aspects about Reasoning about Knowledge*, R. Parikh, Ed., 1990, pp. 97–109.

[38] G. Smith, "A new type system for secure information flow", in *14th Computer Security Foundations Workshop*. IEEE Computer Society, 2001, pp. 115–125.

[39] Sun Microsystems: "Java 2 platform micro edition technology for creating mobile devices". White paper. [Online]. Available: http://java.sun.com/products/cldc/wp/KVMwp.pdf.

Rajeev Prabhakar Goré completed a B.Sc. in physics and computer science and a M.Sc. in design automation at the University of Melbourne, Australia. He obtained his Ph.D. in the proof theory of modal logic from the University of Cambridge, England, in 1991. He was a research associate in computer science at the University of Manchester, England, from 1992-1994. Since then he has been a Research Fellow, Queen Elizabeth II Fellow and a Senior Fellow in Automated Reasoning Group at the Institute of Advanced Studies, Australian National University. His main research interests are in proof theory, automated reasoning and modal logics.

e-mail: rpg@arp.anu.edu.au
Automated Reasoning Group
Computer Sciences Laboratory
and Department of Computer Science
Institute of Advanced Studies and The Faculties
Australian National University
Canberra, ACT 0200, Australia



Phuong Thê Nguyên completed a B.Sc. in computer science at the University of New South Wales, Australia, in 2002. His main research interests include logic and computation.

e-mail: ntp@cs.toronto.edu
Automated Reasoning Group
Computer Sciences Laboratory
and Department of Computer Science
Institute of Advanced Studies and The Faculties
Australian National University
Canberra, ACT 0200, Australia

Invited paper

A formal dynamic semantics of Java: an essential ingredient of Java security

Mourad Debbabi, Nadia Tawbi, and Hamdi Yahyaoui

Abstract — Security is becoming a major issue in our highly networked and computerized era. Malicious code detection is an essential step towards securing the execution of applications in a highly inter-connected context. In this paper, we present a formal definition of Java dynamic semantics. This semantics has been used as a basis to develop efficient, rigorous and provably correct static analysis tools and a certifying compiler aimed to detect and prevent the presence of malicious code in Java applications. We propose a small step operational semantics of a large subset for Java. The latter includes features that have not been completely addressed in the related work or addressed in another semantics style. We provide a fully-fledged semantic handling of exceptions, reachable statements, modifiers and class initialization.

Keywords — security, static analysis, certifying compilers, Java, dynamic semantics, operational semantics, small step semantics.

1. Motivation

Security vulnerabilities occur at two levels, the communication level and the application level. At the communication level the use of cryptographic protocols aims at protecting from security breaches. This issue is being actively studied in order to ensure the absence of flaws in such protocols [1, 7, 11–13]. At the application level, malicious code could be inserted in applications without the consent nor the knowledge of end users. This malicious code can cause data corruption, data divulgation to non authorized users, an extensive use of system resources leading to denial of service, etc. Existing techniques to detect such a code are ad-hoc techniques based on merely syntactic analysis to detect the so-called virus signatures. They can only be effective in the detection of well known and cataloged malicious code. In our research group, we explore three approaches to address malicious code detection: dynamic analysis of code, static analysis of code and self certifying compilation. These approaches are described in [3, 4, 9, 10]. The main idea underlying these approaches is the use of language technology in order to address security issues. In order to make our analysis reliable we base all our techniques on formal and rigorous foundations. For this purpose we elaborated a static semantics for a large subset of the Java language [8]. We present in this paper a dynamic operational small-step semantics for the same Java language subset.

Lately, a surge of interest has been expressed in the elaboration of semantic foundations for Java. This interest is not only motivated by popular appeal and fashion considerations. Indeed, Java has a very sophisticated and subtle semantics as we will exemplify in the sequel. Moreover, Java is meant to be widely used in safety-critical embedded systems. Furthermore, Java support for mobile code through applets poses severe, and very interesting, challenges to the currently established language technologies in terms of security. All these factors justify the need for robust theoretical foundations for Java.

The Java language is, certainly, innovative, but still immature and unstable. Several modifications were made to its description, and errors are still present in its implementations. This is understandable, since the language combines attractive features, which makes its semantics far from being straightforward and leads to substantial complexity.

The only available official specification of Java [15] is an informal description that is subject to different interpretations. Besides, it is rather ambiguous, incomplete and sometimes not consistent with the behavior of the Java compilers. This is not acceptable mainly for the properties that have a direct impact on the security.

We believe that the theoretical investigations of Java semantics are very useful to clarify, correct and complete its semantics description. It will lead, without any doubts, to a better understanding of the language, to a more efficient, safe and secure execution. We strongly believe that a semantics theory for Java is not a luxury but rather a necessity.

A static semantics description has been elaborated in our research group and presented in [8]. In the present paper, we present a dynamic semantics for the same subset.

We believe that the operational semantics style is easy to understand and to manipulate. Actually, the operational style does not require complex mathematical tools which would increase the difficulty to understand the Java language.

Our ultimate goal is to provide a complete formal and easy to use description of the semantic aspects of Java. This will include static as well as dynamic semantics. Another goal we would like to achieve is to prove the subject reduction which guarantees the correctness of our static and dynamic semantic descriptions. This would also be a guarantee that the language is correctly designed i.e. the program behavior is consistent with the typing specification. On the other hand the two semantics could be used as guidelines to ensure a correct design of the Java Virtual Machine (JVM).

The paper is structured as follows. Related work is depicted in Section 2. An evaluation of the semantic issues related to the Java language specification is given in Section 3. A short overview of the Java dynamic semantics is presented in Section 4. Some concluding remarks are ultimately sketched in Section 5 together with some directions for the future work. The syntax of the language and the whole semantic rules set are given in an Appendix.

2. Related work

Many investigations for studying the Java language yielded very interesting results despite the restrictions that have been adopted.

In a pioneering exploration of Java formal semantics, Drossopoulou *et al.* [14] have studied a subset of Java that includes many features like hiding, overloading and exceptions. They proposed an operational semantics for this subset. In order to formalize the evaluation of exceptions, it is not obvious to define rules that could represent the control flow discontinuity which occurs when an exception is raised. The solution proposed by the authors is based on the notion of context. A context encompasses all the enclosing terms up to the nearest enclosing try-catch or try-catch-finally clause i.e. up to the first possible position at which the exception might be handled. Other important features like modifiers and initialization still need to be formalized. Among the assumptions used in [14], the execution of a return statement always terminates the execution of a method. Actually, sometimes we need to execute the enclosing finally clauses before returning to the caller. This adds significant complexity when exceptions are considered in the semantics formalization.

Syme [22] has studied a similar subset, except that he has included, in addition, the local variables. He used the theorem prover *Declare* in order to validate the elaborated operational semantics. The validation consists in proving the soundness of the dynamic semantics w.r.t. the static semantics which he has elaborated. In this work the try-catch, statement is not considered.

Tobias and Oheimb [18] have designed an operational semantics for a subset of Java called *Bali*. They adopted a big-step natural semantics style for elaborating this semantics. Many features of the Java language are considered such as exceptions, local variables, etc. However, the authors did not describe all the possibilities when handling the finally clause. For instance, they did not consider the case where a return statement occurs before the finally clause is evaluated.

Boerger and Sculte [5] have elaborated a dynamic semantics of Java by providing an ASM (abstract state machine) that interprets arbitrary Java programs. They have considered a subset of Java including initialization, exceptions and threads. They have exhibited some weaknesses in the initialization process as far as the threads are used. They pointed out that deadlocks could occur in such a situation.

One of the related work covering almost all Java language is the work of Alves-Foss *et al.* [2]. Their semantics covers the full range of this language excluding concurrency and the Java APIs. This semantics does not address the modifiers. Indeed, the evaluation of a field access expression does not show the modifiers role. Another interesting work is [16]. In this work the author presents a full treatment of the exception mechanism. He uses coalgebras in order to formalize the exception semantics in Java. In [23] the author extends the work done within the *Bali* project to cover exception handling and class initialization. The extension is elaborated in an axiomatic approach. Cenciarelli *et al.* [6] have presented an operational semantics for a significant subset of Java including threads. The major goal of their work is to deal with shared memory.

The operational small step style [19] we have adopted to formalize the dynamic semantics is easy to manipulate and to understand than the denotational and axiomatic styles. It is mandatory to understand the semantics of a language in order to design reliable applications. A precise, formal and easy to understand semantics specification helps to achieve this goal. In our work, we put to the treatment of the exceptions, the class initialization and the modifiers.

3. Semantic issues

The elaboration of a dynamic semantics for Java is a complex task. This complexity is due to many semantic issues related to the Java language. In the sequel, we highlight some of these issues.

3.1. Specification evaluation

The first task when elaborating a formal semantics for any language is to understand the existent informal specification of this language and to evaluate it in order to check whether it is consistent w.r.t. existing compiler reference implementations. Hence, we have started this work by the evaluation of the Java language as it is officially specified in [15]. We have discovered an inconsistency between the aforementioned specification and the JDK 1.3.0 compiler under Linux Redhat 6.2(build Linux_JDK_1.3.0_FCS). It concerns the class initialization process when triggered by a field access expression. Indeed, in the last clarification published by SUN Microsystems in [17], null is considered as a constant expression so every static and final field that is initialized to null is considered as a constant field [15] and cannot trigger class initialization. Actually, access to such a field causes the initialization of the class in which this field is declared.

This inconsistency pointed out that it is very hard to grasp all the subtleties of Java semantics. This is especially crucial when designing a compiler for the language.

One can draw two conclusions. First, the language designers would gain if they adopt a less complex semantics. Second, it is essential to have a non ambiguous specification of the language which is easily understandable.

Table 1
Exception handling

finally

$$\frac{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, Block) \rightarrow (\xi, \mathcal{F}, h') \\ \mathcal{P} = \mathsf{FinalPosition}(Block) \\ \mathcal{F}.ReturnValue = \bot\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{finally}\ \ Block) \rightarrow (\xi, \mathcal{F}, h', \texttt{throw}\ \ \xi, \mathcal{P})}$$

$$\frac{\begin{array}{c}\Gamma \vdash (\mathcal{F}, h, Block) \rightarrow (\mathcal{F}, h') \\ \mathcal{P} = \mathsf{FinalPosition}(Block) \\ \mathcal{F}.ReturnValue \neq \bot \\ \neg\mathsf{HandlerInTable}(Any, \mathcal{F}.Method.ExceptionTable, H, \mathcal{P})\end{array}}{\Gamma \vdash (\mathcal{F}, h, \texttt{finally}\ \ Block) \rightarrow (\mathcal{F}.PreviousFrame, h', \mathcal{F}.ReturnValue)}$$

Table 2
Return statement evaluation

$\boxed{ReturnStatement}$ ....

$$\frac{\neg\mathsf{HandlerInTable}(Any, \mathcal{F}.Method.ExceptionTable, H, \mathcal{P})}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{return}\ v, \mathcal{P}) \rightarrow (\mathcal{F}.PreviousFrame, h, v)}$$

3.2. Java semantics is complex

Java is a real-life language that comes with many convenient features that make its use appealing for the users. The elaboration of a formal semantics for any significant subset of it would be a complex task. The mechanisms underlying some of the interesting features of Java contribute to the complexity of the language semantics. As an example of such mechanisms we can cite the exceptions and the class initialization.

We give, in the sequel, details of the main difficulties or subtleties we encountered while elaborating our dynamic semantics.

The exception mechanism in Java has more complex semantics than other exception mechanisms in other languages due to the finally construct it offers. Actually, a try-catch statement in Java is designed for handling the exceptions that can occur during execution. The try clause contains a block of statements that can raise exceptions. A catch clause can handle an exception and then the execution continues normally. A finally clause may appear in the try statement. This clause is executed whether an exception has occurred or not. A try clause can be enclosed in another one making the semantics more complex especially when a return could occur.

The semantics of jumps and exceptions are usually elaborated using continuation-based techniques [20, 21] in order to preserve the compositionality of the semantic description. Compositionality might be affected by the discontinuity of the control flow caused by jumps or exceptions. Hence a special care is needed.

A continuation models the rest of the code to be executed and is used as a parameter in the semantic functions (denotational framework) or in the semantic rules (operational framework). For the sake of clarity, we use in our semantic rules exception tables that help to compute the needed continuation[1].

Table 1 shows two of the most important rules for the evaluation of the finally clause. A finally clause can be evaluated in the context of an exception ξ. If this clause does not raise any exception and if there is no return statement executed yet then it re-throws the exception ξ. Another interesting rule shows how the execution must return to the caller method after a finally clause is executed. Actually, if there is a return that has been executed and if there is no

[1]The interested reader can refer to the Appendix for more details.

enclosing finally (the predicate HandlerInTable is false), the execution must return to the calling method after returning the resulting value of the return statement evaluation. This is specified in the second rule of Table 1. The previous discussion shows how complex is the handling of a finally clause. In fact its semantics depends on whether a return occurs or not, whether there is another enclosing try or not and whether there is a current raised exception or not.
The rules corresponding to the try and catch clauses evaluation can be found in the Appendix.

3.3. Staged semantics elaboration and its adequacy

Elaborating a dynamic semantics for a reduced subset of Java cannot provide a full understanding of the specifications. When this subset is extended to include the omitted features, a revision of the established rules is often necessary. The dependence of a construct semantics on other constructs makes a Java staged semantics an inadequate strategy. For example, to formalize the semantics of a return statement, it is unavoidable to take in consideration the finally statement as specified in [15]. The rule of the Table 2 shows how the semantics of a return depends on the existence of an enclosing finally. In fact, if there is no enclosing finally to this return the execution returns to the calling method. The statements that appear after the return statement are not necessarily unreachable.

3.4. A deep understanding of the static semantics

Another complexity that we have encountered is that one cannot elaborate a dynamic semantics without understanding the static one. The class initialization illustrates such dependency. Actually, a field access expression can trigger the class initialization if the field is not constant. A constant field is a final, static and initialized by a constant expression at compile time. This information is defined by the static semantics.
Another example concerns the method invocation semantics. In fact, to determine the actual invoked method, a dynamic search process is needed. This search process is based on the method signature which is determined by the static semantics.

4. Short overview of the Java dynamic semantics

In the sequel, we present some aspects of the Java dynamic semantics that we have elaborated.

4.1. Grammar of the Java subset

The syntax of the Java subset that we have considered is given in Tables 13, 14, 15 and 16 of the Appendix. This syntax has been defined in [8]. Notice that this subset is a large one.

4.2. Environment

A dynamic environment is denoted Γ. It consists of a set of class file representations. Each class or interface representation is composed of a set of fields including methods, fields and the *ConstantPool* of the classfile. The complete description of this environment is given in Tables 3 and 4.

4.3. Annotations

The execution of a Java program requires that the compiler adds some relevant information. For instance, a field access would be annotated with the descriptor of this field and the class where it is defined.
Hereafter, we describe the annotations that have been added in the corresponding cases.

Field access expression. Each field access expression *e.f* is annotated with C, the class where this field f is declared, and D its type (descriptor in the Java terminology).

Method invocation. Each method invocation $e.m()$ is annotated with D the method descriptor[2] and C the class where it is declared.

Constructor invocation. An explicit constructor invocation or new instance creation expression is annotated with its descriptor D.

Annotated syntax. We present in Table 5 the syntax annotations which correspond to what a Java compiler generates. The actual syntax that we adopt while elaborating the semantics rules is actually the previously described syntax in which the annotations in the Table 5 are assumed to be propagated.

4.4. Notations

Notations:

- Given two sets A and B, $A \xrightarrow{\sim} B$ denotes the set of all maps from A to B. A map $m \in A \xrightarrow{\sim} B$ could be defined by extension as $[a_0 \mapsto b_0 \ldots a_{n-1} \mapsto b_{n-1}]$ to denote the association of the elements b_i's to a_i's, $a_i \in A$ et $b_i \in B$.
- $dom(m)$ denotes the domain of the map m. Given two maps m and m', we will write $m \dagger m'$ the overwriting of the map m by the associations of the map m' i.e. the domain of $m \dagger m'$ is $dom(m) \cup dom(m')$ and we have $(m \dagger m')(a) = m'(a)$ if $a \in dom(m')$ and $m(a)$ otherwise.
- $s \oplus s'$ denotes the disjoint union of the two sets s and s'.
- $S[f \leftarrow v]$ denotes the assignment of the value v to the field f of the structure S.
- ϵ denotes an empty value, cf Section 4.6.

[2]Descriptor stands for signature in Java terminology.

Table 3
Environment – Part 1

Environment	::=	(*ClassFile*)set				
ClassFile	::=	(	*ConstantPool*	:	(*CpInfo*)set	
			Modifiers	:	(*ClassModifier*)set	
			ThisClass	:	*ClassType*	
			SuperClass	:	*ClassType*	
			Interfaces	:	(*ClassType*)set	
			Fields	:	(*FieldInfo*)set	
			Methods	:	(*MethodInfo*)set	
			Initialised	:	*boolean*	)
FieldInfo	::=	(	*Modifiers*	:	(*FieldModifier*)set	
			SimpleName	:	String	
			Descriptor	:	*FieldDescriptor*	
			Constant	:	*boolean*	)
MethodInfo	::=	(	*Modifiers*	:	(*MethodModifier*)set	
			SimpleName	:	String	
			Descriptor	:	*MethodDescriptor*	
			Code	:	(*Statement*)list	
			ExceptionTable	:	(*ExceptionHandler*)list	
			LocalVariableTable	:	Identifier $\xrightarrow{\sim}$ *Value*	)
ClassModifier	::=	public | static | interface | final | private				
FieldModifier	::=	public | static | protected | final | private | transient | volatile				
MethodModifier	::=	public | static | abstract | final | private | protected | synchronized | native | transient | volatile				
CpInfo	::=	*FieldrefInfo*				
	|	*MethodrefInfo*				
	|	*InterfaceMethodrefInfo*				
FieldrefInfo	::=	(	*FromClass*	:	*ClassFile*	
			NameAndType	:	*NameAndTypeInfo*	)
NameAndTypeInfo	::=	(	*Name*	:	String	
			Descriptor	:	*FieldDescriptor*	
				|	*MethodDescriptor*	)
ExceptionHandler	::=	(	*From*	:	integer	
			To	:	integer	
			Target	:	*Statement*	
			Type	:	*ExceptionType*	)

Table 4
Environment – Part 2

ExceptionType	::=	*ClassFile*
	|	*Any*
Any	::=	*ClassType*
	|	*ArrayType*
FieldDescriptor	::=	*FieldType*
FieldType	::=	*PrimitiveType*
	|	*ClassType*
	|	*ArrayType*
PrimitiveType	::=	`byte` | `char` | `double` | `float`
		`integer` | `long` | `short` | *boolean*
ArrayType	::=	*SimpleType*[]
SimpleType	::=	*PrimitiveType*
	|	*ClassOrInterfaceType*
ClassOrInterfaceType	::=	*ClassType*
	|	*InterfaceType*
ClassType	::=	*ClassFile*
InterfaceType	::=	*ClassFile*
MethodDescriptor	::=	(*ParameterDescriptor*) *ReturnDescriptor*
ParameterDescriptor	::=	*FieldType*
ReturnDescriptor	::=	*FieldType*
	|	`void`

Table 5
Annotations

ExplicitConsInvocation	::=	[*D*] `this` (*Argument*) ;
	|	[*D*] `super`(*Argument*) ;
	|	ε
ClassInstanceCreation	::=	[*D*] `new` *ClassType* (*Argument*)
SimpleFieldAccess	::=	*Primary* . [*C*, *D*] `Identifier`
	|	`super` . [*C*, *D*] `Identifier`
	|	[*C*, *D*]*FieldName*
MethodInvocation	::=	[*C*, *D*]*MethodName* (*Argument*)
	|	*Primary* .[*C*, *D*] `Identifier` (*Argument*)
	|	`super` .[*C*, *D*] `Identifier` (*Argument*)

Table 6
Configurations

$$\begin{array}{l}
\mathit{Configuration} \quad ::= (\mathit{ExceptionObject}\ \ \mathit{option},\ \mathit{Frame},\ \mathit{heap},\ \mathit{IntermediateTerm},\\
\qquad\qquad\qquad\quad \mathit{Position}\ \ \mathit{option})
\end{array}$$

$$\begin{array}{lllll}
\mathit{Frame} & ::= & \{ & \mathtt{this} & : & \mathit{RefValue} \\
 & & & \mathit{Class} & : & \mathit{ClassFile} \\
 & & & \mathit{Method} & : & \mathit{MethodInfo} \\
 & & & \mathit{PreviousFrame} & : & \mathit{Frame} \\
 & & & \mathit{ReturnAddress} & : & \mathtt{int32} \\
 & & & \mathit{ReturnValue} & : & \mathit{Value}\ \ \}
\end{array}$$

$$\begin{array}{lllll}
\mathit{heap} & : & \mathit{FieldRecord} & \xrightarrow{\sim} & \mathit{Value} \\
 & \sqcup & \mathtt{address} & \xrightarrow{\sim} & (\mathit{ClassFile},\ \mathit{MF}) \mid (\mathit{ClassFile},\ \mathit{MI})
\end{array}$$

$$\mathit{MF} \quad : \quad \mathit{FieldRecord} \quad \xrightarrow{\sim} \quad \mathit{Value}$$

$$\mathit{MI} \quad : \quad \mathtt{int32} \quad \xrightarrow{\sim} \quad \mathit{Value}$$

$$\begin{array}{llllll}
\mathit{FieldRecord} & ::= & \{ & \mathit{Field} & : & \mathit{FieldInfo} \\
 & & & \mathit{FromClass} & : & \mathit{ClassFile}\ \ \}
\end{array}$$

$$\mathit{ExceptionObject} \quad ::= \quad \mathtt{address}$$

$$\mathit{Position} \quad ::= \quad \mathtt{int32}$$

Table 7
Computable values

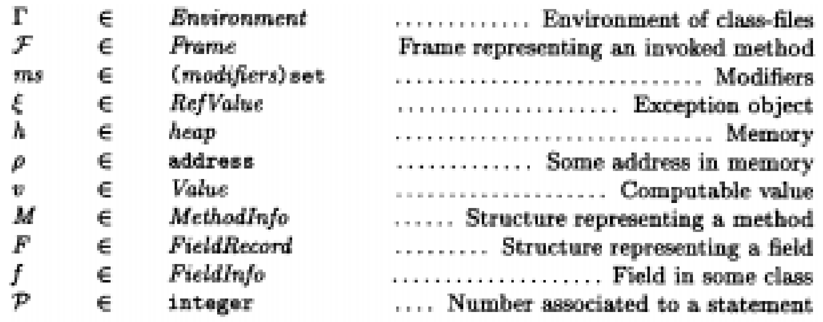
$$\begin{array}{lll}
\mathit{Value} & ::= & \mathit{PrimValue} \\
 & \mid & \mathit{RefValue} \\
 & \mid & \mathtt{null} \\
 & \mid & \bot \\
 & \mid & (\mathit{Value})\,\mathtt{set}
\end{array}$$

$$\begin{array}{lll}
\mathit{PrimValue} & ::= & \mathtt{byte8} \oplus \mathtt{short16} \oplus \mathtt{int32} \oplus \mathtt{long64} \\
 & & \oplus\ \mathtt{char16} \oplus \mathtt{iee32} \oplus \mathtt{iee64} \oplus \mathit{boolean}
\end{array}$$

$$\mathit{RefValue} \quad ::= \quad \mathtt{address}$$

$$\mathit{boolean} \quad ::= \quad \mathtt{true} \oplus \mathtt{false}$$

Table 8
Semantic categories



Γ	$\in$	*Environment*	.... Environment of class-files
$\mathcal{F}$	$\in$	*Frame*	Frame representing an invoked method
ms	$\in$	*(modifiers)* `set`	.... Modifiers
ξ	$\in$	*RefValue*	.... Exception object
h	$\in$	*heap*	.... Memory
ρ	$\in$	`address`	.... Some address in memory
v	$\in$	*Value*	.... Computable value
M	$\in$	*MethodInfo*	...... Structure representing a method
F	$\in$	*FieldRecord*	.... Structure representing a field
f	$\in$	*FieldInfo*	.... Field in some class
$\mathcal{P}$	$\in$	`integer`	.... Number associated to a statement

4.5. Intermediate terms

Since we adopted a small step style, we have to represent intermediate results during the evaluation process. These intermediate results are formalized as algebra terms and are denoted intermediate terms. Actually, *intermediate terms* consist of terms that involve syntactic entities as well as computable values. For instance, let $T[e]$ be a Java expression where T is an array and e is a Java expression. The evaluation of this expression yields as an intermediate result $T[v]$ where v stands for a computable value representing the result of the evaluation of e. Clearly, this intermediate result does not belong to the Java syntax because the integer values are different from the integer constants that would appear in a Java source code.

4.6. Configurations

A configuration is a tuple ($\xi, \mathcal{F}$, h, t) where ξ is the object exception that is thrown and not yet handled, $\mathcal{F}$ is the frame of the current method, h is the global memory and t is an intermediate term. For the sake of convenience, this configuration may appear as ($\mathcal{F}$, h, t) when no exception is thrown and not handled. Actually, this is an abbreviation of (ξ, $\mathcal{F}$, h, t) where ξ is empty, denoting the absence of exceptions. The configuration may also appear as ($\mathcal{F}$, h) where there is no thrown-exception after evaluating the term t and this term is fully evaluated. The Table 6 shows the configurations used in our semantics. In our configurations the notation ξ^+ represent either an exception ξ or the absence of a raised exception denoted by ϵ.

4.7. Computable values

The evaluation of the syntactic constructs of a language using a formal semantics produces values that are commonly denoted *computable values*. We define in Table 7 the computable values manipulated by our dynamic semantics. A computable value can be a primitive value or a reference (memory address). We introduce the undefined value ($\perp$) which is used as:

- the value of this in a static method,
- the value of the field *ReturnValue* in the current frame if no return statement has been executed yet.

4.8. Memory abstraction

We abstract the memory by a map h which associates a computable value to a *RefValue* (cf Table 7) or to a *FieldRecord* which represents a class field (cf Table 6). In the map h a *FieldRecord* always corresponds to a static field. Notice that a *RefValue* represents a reference to an object or to an array.
The following items describe how objects and arrays are represented in our memory abstraction:

- An object is represented by an ordered pair (*ClassType, MF*) where $MF = [F_0 \mapsto v_0, \dots, F_{n-i} \mapsto v_{n-i}]$, *ClassType* is the concrete type of the object, F_i is a *FieldRecord* structure corresponding to a non static field and v_i represents the computable value associated to this field.
- An array T is represented by an ordered pair (*ClassType, MI*) where $MI = [0 \mapsto v_0, \dots, n-1 \mapsto v_{n-1}]$, n is the dimension of the array and v_i is a computable value associated with $T[i]$.

4.9. Semantic categories

We define in Table 8 the semantic categories manipulated by our semantics.

4.10. Semantic rules

The evaluation process is formalized as a transformation of a configuration to a new one. We denote this transforma-

Table 9
Return statement evaluation

$\boxed{\mathit{ReturnStatement}}$ ....

$$\frac{\begin{array}{c}\textsf{HandlerInTable}(\mathit{Any}, \mathcal{F}.\mathit{Method}.\mathit{ExceptionTable}, H, \mathcal{P}) \\ \mathit{Statement} = H.\mathit{Target}\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{return}\ v, \mathcal{P}) \rightarrow (\mathcal{F}[\mathit{ReturnValue} \leftarrow v], h, \mathit{Statement})}$$

$$\frac{\neg\textsf{HandlerInTable}(\mathit{Any}, \mathcal{F}.\mathit{Method}.\mathit{ExceptionTable}, H, \mathcal{P})}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{return}\ v, \mathcal{P}) \rightarrow (\mathcal{F}.\mathit{PreviousFrame}, h, v)}$$

Table 10
Exception handling

$\boxed{\mathit{finally}}$ ....

$$\frac{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{Block}) \rightarrow (\xi, \mathcal{F}, h') \\ \mathcal{P} = \textsf{FinalPosition}(\mathit{Block}) \\ \mathcal{F}.\mathit{ReturnValue} = \perp\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{finally}\ \ \mathit{Block}) \rightarrow (\xi, \mathcal{F}, h', \texttt{throw}\ \ \xi, \mathcal{P})}$$

$$\frac{\begin{array}{c}\Gamma \vdash (\mathcal{F}, h, \mathit{Block}) \rightarrow (\mathcal{F}, h') \\ \mathcal{P} = \textsf{FinalPosition}(\mathit{Block}) \\ \mathcal{F}.\mathit{ReturnValue} \neq \perp \\ \neg\textsf{HandlerInTable}(\mathit{Any}, \mathcal{F}.\mathit{Method}.\mathit{ExceptionTable}, H, \mathcal{P})\end{array}}{\Gamma \vdash (\mathcal{F}, h, \texttt{finally}\ \ \mathit{Block}) \rightarrow (\mathcal{F}.\mathit{PreviousFrame}, h', \mathcal{F}.\mathit{ReturnValue})}$$

tion by $\Gamma \vdash (\xi, \mathcal{F}, h, t) \rightarrow (\xi', \mathcal{F}', h', t')$ which says that an intermediate term t is evaluated under the exception ξ, in the frame $\mathcal{F}$. The result of this evaluation is the new intermediate term t'. The evaluation may modify the current memory h, may raise a new exception and finally may change the current method being evaluated. ξ' stands for the new exception, $\mathcal{F}'$ stands for the new frame corresponding to the new method and h' stands for the new memory. The operational semantics consists of a set of semantic rules. Each rule states that the evaluation in the conclusion part can be deduced from the evaluations in the premise part.
The complete set of the semantic rules is given in the Appendix. We give in the sequel the explanation of some relevant rules. The remaining rules in the Appendix should be understood in a similar way.

4.10.1. The return statement

The evaluation of a return statement is very subtle. Actually, after the execution of a return statement, every enclosing finally clause must be executed. A predicate HandlerInTable (cf Section A.3.5 in the Appendix) indicates if a finally clause exists in the exceptions table of the method represented by the frame $\mathcal{F}$. By the number associated to this return statement, we can get the first enclosing finally, if it exists, in the exceptions table of the method in $\mathcal{F}$ (cf Tables 26 and 27). *H. Target* represents this finally statement. The value returned by the evaluation is assigned to the field *ReturnValue* of the frame $\mathcal{F}$ representing the current method, the current exception is given up and the execution continues by handling the *Block* of the finally clause. This is specified in the first rule in Table 9.
When there is no enclosing finally clause, the execution continues in the calling method by returning the value of the return evaluation. This is specified in the second rule in Table 9.

4.10.2. Exceptions

Exception handling in Java is a highly designed mechanism that provides to developers the possibility to deal with abnormal situations without causing the execution abortion. Actually, the developer can control bad results and associate a specific code to handle such situations. A try statement in Java is designed for handling exceptions that can occur through execution. The try clause contains a block of

Table 11
Static method call evaluation

$\boxed{\mathit{MethodInvocation}}$ ....

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{Argument}) \rightarrow (\xi', \mathcal{F}', h', \mathit{Argument}')}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{ClassType}.[C, D]\mathtt{Identifier}(\mathit{Argument})) \rightarrow \\ (\xi', \mathcal{F}', h', \mathit{ClassType}.[C, D]\mathtt{Identifier}(\mathit{Argument}'))\end{array}}$$

$$\frac{\begin{array}{c}\mathsf{InMethod}\ (M, \mathtt{Identifier}, C, D) \\ \mathtt{static} \in M.\mathit{Modifiers} \\ \mathtt{Id} = \mathsf{GetFormalParameter}(M) \\ \mathcal{F}' = (\bot, \mathit{ClassType}, M, \mathcal{F}, \mathcal{P}, \bot) \\ \mathcal{LV} = \mathcal{F}'.\mathit{Method.LocalVariableTable} \dagger [\mathtt{Id} \mapsto v] \\ \neg\mathsf{initialized}(C)\end{array}}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{ClassType}.[C, D]\mathtt{Identifier}(v), \mathcal{P}) \rightarrow \\ (\xi, \mathcal{F}'[\mathit{Method.LocalVariableTable} \leftarrow \mathcal{LV}], h, C.[C, ()\mathtt{void}]\mathtt{clinit}()\ ; \\ \mathcal{F}'.\mathit{Method.Code})\end{array}}$$

Table 12
Instance method call evaluation

$\boxed{\mathit{MethodInvocation}}$ ....

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{Argument}) \rightarrow (\xi', \mathcal{F}', h', \mathit{Argument}')}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, [C, D]\mathtt{Identifier}(\mathit{Argument})) \rightarrow \\ (\xi', \mathcal{F}', h', [C, D]\mathtt{Identifier}(\mathit{Argument}'))\end{array}}$$

$$\frac{\begin{array}{c}\mathsf{InMethod}\ (M, \mathtt{Identifier}, C, D) \\ \mathtt{static} \notin M.\mathit{Modifiers} \quad \mathtt{private} \notin M.\mathit{Modifiers} \\ \mathit{InvocMode} = \mathsf{GetInvocMode}(M, \mathtt{false}) \\ S = \mathsf{ConcreteType}(\mathcal{F}.\mathtt{this}) \\ (M', R) = \mathsf{LookupFirstSuperClass}(\mathcal{F}.\mathit{this}, M, S, \mathit{InvocMode}) \\ \mathtt{Id} = \mathsf{GetFormalParameter}(M') \\ \mathcal{F}' = (\mathcal{F}.\mathtt{this}, R, M', \mathcal{F}, \mathcal{P}, \bot) \\ \mathcal{LV} = \mathcal{F}'.\mathit{Method.LocalVariableTable} \dagger [\mathtt{Id} \mapsto v]\end{array}}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, [C, D]\mathtt{Identifier}(v), \mathcal{P}) \rightarrow \\ (\xi, \mathcal{F}'[\mathit{Method.LocalVariableTable} \leftarrow \mathcal{LV}], h, \mathcal{F}'.\mathit{Method.Code})\end{array}}$$

statements that can raise exceptions. A catch clause can handle an exception and then the execution continues normally. A finally clause may appear in the try statement. This clause is executed whether an exception has occurred or not. It is considered as a clean code. We present two rules for the finally evaluation. The rules corresponding to the try and catch clauses evaluation can be found in the Appendix).

Table 10 shows two rules for the evaluation of the finally clause. A finally clause can be evaluated in the context of an exception ξ. If this clause does not raise any if exception and there is no return statement executed yet ($\mathcal{F}.ReturnValue\ = \bot$) then it re-throws the exception ξ at the position of the last statement of finally. The function finalPosition returns the number of the last statement of a block of statements. This is specified in the first rule in Table 10.

Another interesting rule shows how the execution must return to the caller method after a finally clause. Actually, if there is a return that has been executed ($\mathcal{F}.Returnvalue \neq \bot$) and if there is no enclosing finally (the predicate HandlerInTable evaluates to false), the execution must return to the caller method (represented by $\mathcal{F}.PreviousFrame$) after returning the resulting value of the return evaluation. This is specified in the second rule in Table 10.

4.10.3. Method invocation

Java uses the dynamic dispatch to determine which method is to be executed in each call site when an instance method is invoked. The search of the actual method is performed at run time. Actually, the JVM launches a search procedure to determine the method to be executed. On the other hand, if the invoked method is static or private there is no need to launch this search the actual method would be statically determined. The semantic rules corresponding to the method invocation represent this process as a first step of the evaluation.

After determining the method to be invoked, we proceed to the evaluation of the actual parameters. Let M be the invoked method and C the class where it is declared. If static belongs to the modifiers set of M, the evaluation of the invocation must trigger the initialization of the class C if it is not initialized yet. This is checked using the predicate initialized. If C is not initialized we call its clinit() method before evaluating the code of M. The code of M is evaluated under the substitution of the formal parameter by the value of the argument. A new frame is created and used along the evaluation of the method code. The semantic rules of evaluating a static method as well as an instance method are shown in Tables 11 and 12. If the method is an instance many operations are performed before executing its code. Let M be the instance method, C the class where it is declared as determined at compile time and D its descriptor. The first evaluation step, is the search of the actual method. For this, a predicate InMethod checks the existence of a method M having a descriptor D in the class C. The second evaluation step consists in searching for the actual method to be executed at this site. This depends on the type of the receiving object. The semantic function LookupFirstSuperClass performs this search. Notice that we consider methods with one parameter if any. We made this restriction only to seek more clarity of the rules. The generalization to more than one parameter could be easily performed. For more details cf Section A.3.13 in the Appendix.

5. Conclusion and future work

We discussed in this paper a dynamic semantics of a large subset of Java in which we have handled some subtle problems such as initialization, modifiers, and exceptions. The formalization has been carried out in an operational small step style. This makes it extendable to handle another aspects of the language such as the threads. On the other hand, this style is easily understood and manipulated without any heavy theoretical background. The whole semantics is detailed in the Appendix. We plan to extend this semantics to include packages, inner-classes and threads and ultimately to prove the consistency between this semantics and the static one described in [8].

On the other hand, this paper gives some insights into the task consisting in elaborating a Java dynamic semantics. We can sum up our conclusions as follows. Elaborating a dynamic semantics for Java should treat all features of the language not just a reduced subset of it. Hence, a staged semantics strategy would be inadequate. Any research effort that address the whole language or at least a realistic subset of it would be a worthwhile. We hope that SUN Microsystems simplifies some constructs semantics such as the exceptions.

This work contributes to build a formal foundations for our techniques and tools designed to address verification of Java applications properties related to security issues.

References

[1] K. Adi, M. Debbabi, and M. Mejri, "A logic for specifying security properties of electronic commerce protocols", in *Proceedings 8th International Conference on Algebraic Methodology and Software Technology, AMAST'2000*, *Lecture Notes in Computer Science*. Springer, 2000, vol. 1816, pp. 499–513 (also, accepted for publication in the *Int. J. Theor. Comput. Sci.*, Elsevier).

[2] J. Alves-Foss and F. S. Lam, "Dynamic denotational semantics of Java", in *Formal Syntax and Semantics of Java*, J. Alves-Foss, Ed., *LNCS*. Springer, 1999, pp. 201–240.

[3] J. Bergeron, M. Debbabi, J. Desharnais, M. Erhioui, Y. Lavoie, and N. Tawbi, "Static detection of malicious code in executable programs", in *Symp. Requir. Eng. Inform. Secur.*, May 2000.

[4] J. Bergeron, M. Debbabi, J. Desharnais, M. Erhioui, Y. Lavoie, and N. Tawbi, "Static detection of malicious code in executable programs", in *REJ Special Issue Requir. Eng. Inform. Secur.*, June 2001.

[5] E. Boerger and W. Schulte, "A programmer friendly modular definition of the semantics of Java", in *Formal Syntax and Semantics of Java*, J. Alves-Foss, Ed., *LNCS*. Springer, 1999, pp. 353–404.

[6] P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing, "An event-based operational semantics of multi-threaded Java", in *Formal Syntax and Semantics of Java*, J. Alves-Foss, Ed., *LNCS*. Springer, 1999, pp. 157–200.

[7] M. Debbabi, N. Durgin, M. Mejri, and J. Mitchell, "Security by typing", in *Int. J. Softw. Tools Technol. Transfer (STTT)*, 2001.

[8] M. Debbabi and M. Fourati, "Sur la sémantique statique de Java", in *LMO*, Montréal, Jan. 2000, pp. 167–182.

[9] M. Debbabi, E. Giasson, B. Ktari, F. Michaud, and N. Tawbi, "Secure self-certified code", in *Proc. IEEE 9th Int. Worksh. Enterpr. Secur. (WETICE'OO)*, National Institute of Standards and Technology (NIST), Maryland, USA, June 2000.

[10] M. Debbabi, M. Girard, L. Poulin, M. Salois, and N. Tawbi, "Dynamic monitoring of malicious activity in software systems", in *Symp. Requir. Eng. Inform. Secur.*, May 2000.

[11] M. Debbabi, M. Mejri, N. Tawbi, and I. Yahmadi, "A new algorithm for the automatic verification of authentication protocols", in *DIMACS Worksh. Des. Form. Verif. Secur. Protoc.*, Sept. 1997.

[12] M. Debbabi, M. Mejri, N. Tawbi, and I. Yahmadi, "Formal automatic verification of authentication cryptographic protocols", in *First IEEE Int. Conf. Form. Eng. Meth. (ICFEM'97)*, Nov. 1997.

[13] M. Debbabi, M. Mejri, N. Tawbi, and I. Yahmadi, "From protocol specifications to flaws and attack scenarios: an automatic and formal algorithm", in *Worksh. Enterpr. Secur. (WETICE'97)*, June 1997.

[14] S. Drossopoulou, T. Valkevych, and S. Einsenbach, "Java type soundness revisited". Tech. Rep., Imperial College of Science, Technology and Medicine, Oct. 1999.

[15] J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*. Addison Wesley, 1996.

[16] B. Jacobs, "A formalization of Java's exception mechanism", in *Programming Languages and Systems*, D. Sands, Ed., *LNCS*. Springer, 2001, pp. 284–301.

[17] "SUN Microsystems. Clarifications and amendments to the Java language specification", 1998. [Online]. Available: http://java.sun.com/docs/books/jls/clarify.html

[18] T. Nipkov and D. Oheimb, "Machine-checking the Java specification: proving type-safety", in *Formal Syntax and Semantics of Java*, J. Alves-Foss, Ed., *LNCS*. Springer, 1999, pp. 119–156.

[19] G. D. Plotkin, "A structural approach to operational semantics". Tech. Rep. FN-19, DAIMI, University of Aarhus, Denmark, Sept. 1981.

[20] J. C. Reynolds, "The discoveries of continuations", in *Lisp and Symbolic Computation*. 1993, pp. 233–247.

[21] C. Strachey and C. P. Wadsworth, "Continuations a mathematical semantics for handling full jumps". Tech. Rep. PRG-11, Oxford University Computing Laboratory, Programming Research Group, Jan. 1974.

[22] D. Syme, "Proving Java type soundness", in *Formal Syntax and Semantics of Java*, J. Alves-Foss, Ed., *LNCS*. Springer, 1999, pp. 83–118.

[23] D. von Oheimb, "Axiomatic semantics for Java light in Isabelle/HOL", in "Formal techniques for Java programs", S. Drossopoulou, S. Eisenbach, B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, Eds. Tech. Rep. 269, 5/2000, Fernuniversität Hagen, Fernuniversität Hagen, 2000. [Online]. ECOOP2000 Workshop Proc. available: http://www.informatik.fernuni-hagen.de/pi5/publications.html

Appendix
A full overview of the Java dynamic semantics

We present here a complete overview of the dynamic operational semantics that we have elaborated.

A.1. Grammar of the subset

The syntax of the Java subset that we have considered is given in Tables 13, 14, 15 and 16 (Tables 13–48, see this issue, pp. 97–119). This syntax has been defined in [8].

A.2. Hypothesis

The following hypothesis are assumed in our work. We present them together with the rationale underlying their assumption:

- Our semantics is able to evaluate syntactically correct programs. Furthermore, we assume that all the needed classes have been loaded and checked. We also assume that the reference resolution step has been performed correctly. These assumptions are essential. Actually, we do not formalize the dynamic linking process.

- We assume that the Java program is preprocessed so that each statement is identified by a number and each expression is tagged with some relevant annotations. These annotations are described in the sequel. These annotations correspond actually to what the compiler generates.

- For the sake of clarity, we assume that all the methods have only one argument. We also consider that all arrays are mono-dimensional. The generalization is obvious in both cases, but would unnecessarily make cumbersome the presentation.

- We consider that two methods have been added in each class, namely init() and clinit(). These methods have been added in order to express the initialization process as it is performed by the JVM.

  – init (*Argument*): this method represents the constructor code of the class and the initializers of the instance variables. *Argument* is the parameter of the constructor. It can be void.

  – clinit(): this method contains the class static code and the initializers of the static variables.

  It is clear that these two methods represent what actually is generated by the compiler.

A.3. Semantics rules

The evaluation process is formalized as a transformation of a configuration to a new one. We denote this transformation by $\Gamma \vdash (\xi, \mathcal{F}, h, t) \rightarrow (\xi', \mathcal{F}', h', t')$ which means that an intermediate term t is evaluated under the exception ξ, in the frame $\mathcal{F}$. The result of this evaluation is the new intermediate term t'. The evaluation may modify the current memory h, may raise a new exception and finally may change the current method being evaluated. ξ' stands for the new current exception, $\mathcal{F}'$ stands for the new frame corresponding to the new method and h' stands for the new memory.
The operational semantics consists of a set of semantic rules. Each rule states that the evaluation in the conclusion part can be deduced from the evaluations in the premise part.

A.3.1. Field declaration evaluation

The following remarks help the understanding of the field declaration evaluation rules.

- A *FieldDeclaration* expression is considered as a part of clinit() declaration or init() one depending on the modifiers of this field. If the field is static then its declaration will be included in the clinit() method otherwise it will be in the init() method.

- We represent each static field in the memory by a *FieldRecord* that contains the field itself (*FieldInfo*) and the class in which it is declared (*ClassFrom*).

- Each declaration of a static field adds to the memory the *FieldRecord* with its default value or the value resulting from the evaluation of the expression that initializes it.

- An instance field declaration is included in the init() method and will be executed when a new object is created.
- A field is considered as an instance field when static $\notin$ modifiers of this field.
- A *FieldRecord* is added to an object having this as its address.
- The concrete type of an object having ρ as an address is obtained by applying the function ConcreteType(ρ).

The methods clinit() and init() are defined in Tables 17 and 18.
In the field declaration evaluation rules, InField stands for a predicate which evaluates to true when the field represented by the first argument belongs to the class represented by the third argument and false otherwise. The second argument of the predicate InField stands for the simple name of the field. The fourth argument represents the type of the field.

InField : *FieldInfo* × Identifier × *ClassFile* ×
× *FieldDescriptor* → bool

InField $(f$, Identifier, C, $D)$ =
$(f \in C.Fields) \wedge$
$(f.SimpleName =$Identifier$) \wedge$
$(f.Descriptor = D)$.

The rules of field declaration evaluation are presented in Tables 19 and 20.

A.3.2. Constructor evaluation

A constructor invocation (Table 21) is equivalent to the invocation of the method init() of the class that represents the concrete type of the newly created object. For example, an explicit constructor invocation like this (*Argument*) is annotated with its descriptor D as follows [D] this (*Argument*) is evaluated to this.[C, $D|$init(υ) where C is the concrete type of this and υ is the value of *Argument*.

A.3.3. Local variable declaration expression evaluation

A *LocalVarDeclaration* expression (Table 22) adds a new variable to the local environment of the current method. An initialization of a local variable updates its value in the local variable table of the current method.

A.3.4. Statement evaluation

Statement evaluation:

- The statement if-then: first the condition of the if clause is evaluated. When its value is true, the then clause is executed otherwise the configuration is not modified.
- The statement if-then-else: first the condition is evaluated to produce υ as a value. If υ is true then the clause then is evaluated, otherwise the clause else is evaluated.
- The statement while: we evaluate first the condition, when its value is false the configuration does not change, otherwise the evaluation of this statement produces a statement if having the body of the while statement as its then clause.

The evaluations of the previous statements are presented in Tables 23 and 24.
The evaluation of the return statement is presented in Table 25.

A.3.5. Exception handling

We suppose that a preprocessing of each method is performed in order to associate numbers with statements. These numbers respect the textual order. The exceptions table indicates where the control has to flow (continuation) after each potential exception occurrence in a try-catch construct. For example, the exceptions Table 27 is associated with the piece of code in Table 26. The column *Target* represents the statement block of a catch that can handle an exception in the clause try which is thrown between statement 1 and statement *i*. The block of the finally clause is executed if the exception is thrown between statement 1 and *n* whether a catch has been executed or not, where *n* represents the number of the last statement in the last catch.
A throw statement raises an exception from the position where it appears. First *Argument*, the argument of the throw, is evaluated. It produces a reference value υ. If υ is null then a NullPointerException is raised at the same position as the throw statement. Otherwise, we must search a handler for the thrown exception E. This is formalized by the predicate HandlerlnTable($E, H, \mathscr{F}.Method.ExceptionTable, \mathscr{P}$) which evaluates to true if the first enclosing catch or the first enclosing finally exists in the exceptions table given as its third argument. This exceptions table is associated with the current frame $\mathscr{F}$. The predicate evaluates to false otherwise.
A try statement can have one of the following three forms: try-catch or try-finally or try-catch-finally.
A try clause is a guarded section where each abnormal execution will cause a jump to a catch which argument type is a supertype of the raised exception type.
A finally clause is known as a clean code that will be executed whether a previous exception has occurred or not. When a finally clause exists in a try statement, the program must execute this clause whether an exception occurred or not in the associated try and/or catch clauses of the same statement. Accordingly, we should formally state such a semantic constraint. In the exception table, we specify the column *ExceptionType* which contains the type of the thrown exception. We introduce the type *Any* representing every non-primitive type that can exist. Hence, if an

exception is thrown, it will verify the constraint of subtyping with *Any*. The example of Table 26 shows a try statement with a finally clause: When an exception is thrown at some position $\mathscr{P}$, the program execution continues at the first *Target* in the table that corresponds to a type in the column Type, which is a supertype of this exception. We define the predicate HandlerlnTable that checks the existence of the first catch or finally in the exceptions table of the method represented by the frame $\mathscr{F}$ i.e. the first range (*From-To*) that contains the position $\mathscr{P}$ where an exception E has occurred and having an exception type in the third column as a superclass of E (this relation is defined under the environment Γ). It returns the *Block* contained in the *Target* column. This *Block* represents the continuation of the execution after the occurrence of the exception. The predicate HandlerlnTable is defined as follows:

HandlerlnTable : $\mathit{ExceptionType} \times \mathit{ExceptionTable} \times$
$\times\ \mathit{ExceptionHandler} \times$ Position $\rightarrow$ bool

$$\begin{array}{l}
\text{HandlerlnTable}(E, \mathscr{F}.Method.ExceptionTable, H, \mathscr{P}) = \\
(\mathscr{F}.Method.ExceptionTable = [\,] \Rightarrow \text{false}) \vee \\
((H = \text{hd}(\mathscr{F}.Method.ExceptionTable)) \wedge \\
(\mathscr{P} \geq H.From) \wedge \\
(\mathscr{P} \leq H.To) \wedge \\
((H.Type = Any) \vee ((E \sqsubseteq H.Type) \wedge (E \neq Any)))) \vee \\
((\text{HandlerlnTable}(E, \text{tl}(\mathscr{F}.Method.ExceptionTable), \\
H, \mathscr{P})) \wedge \\
(\mathscr{P} < \text{hd}(\mathscr{F}.Method.ExceptionTable).From) \vee \\
(\mathscr{P} > \text{hd}(\mathscr{F}.Method.ExceptionTable).To) \vee \\
((\text{hd}(\mathscr{F}.Method.ExceptionTable).Type) \neq Any) \wedge \\
((E \not\sqsubseteq (\text{hd}(\mathscr{F}.Method.ExceptionTable).Type)) \vee \\
(E = Any)))
\end{array}$$

Let us explain now some rules from Tables 28, 29 and 30:

- A try clause can be executed without raising any exception. If there is an enclosing finally statement in the exceptions table, the execution will continue at the *Block* of this finally, otherwise it will continue normally. If this clause raises an exception, it will be equivalent to a throw statement at the position where the exception occurred.

- A catch clause is executed when it handles a thrown exception that has happened before (and not from another catch clauses in the same statement). So, no exception will be present in the configurations before executing this clause. The enclosing finally clause is then determined and the execution continues at the *Block* of this finally if it is found. When this catch clause raises an exception it will be equivalent to a throw statement at the position of the statement that has caused it.

- The finally clause is more complicated to formalize. In fact, there are two factors that influence the evaluation: first, a return statement (if it has been executed before this finally or inside it) and second, if there is an exception before executing it or caused by this clause itself. When the evaluation of this clause terminates normally and there is no return statement that has been executed ($\mathscr{F}.ReturnValue = \perp$ where $\mathscr{F}$ is the current frame) and no other enclosing finally (the predicate Handlerlntable evaluates to false), the execution continues normally. When a return has been executed before this finally and there is no thrown exception in the left configuration, we must go to the first enclosing finally if it exists. Otherwise the execution returns to the calling method. The execution of finally can itself raise an exception and it will be equivalent in this case to a throw. When a finally clause is executed under some exception and raises by itself another exception, it gives up the former and raises this new exception. If it does not cause another exception it removes the initial exception.

A.3.6. New array creation expression evaluation

An array creation expression returns a new reference to the created array. Each element of the array is initialized by its default value. A NegativeSizeException is raised when the array size is negative. The Table 31 shows the semantics of such an expression.

A.3.7. Literal, this and parenthesi ed expression evaluation

A literal is evaluated to its primitive value this is evaluated to the field this of the current frame. Evaluation of a parenthesized expression returns the value of the expression that is inside the parentheses. All these rules are formally stated in the Table 32.

A.3.8. New class instance creation expression evaluation

The Table 33 states how to evaluate a new instance class creation which triggers a call of the method init() and the initialization of this class (call of clinit() if it is not yet initialized). To obtain all the fields of some class C, we use the function Fields(C).

Fields : $\mathit{ClassFile} \rightarrow (\mathit{FieldRecord})$set

$$\begin{array}{ll}
\multicolumn{2}{l}{\text{Fields } (C) =} \\
\forall & F \in C.Fields. \\
 & \text{if static} \notin F.Modifiers \\
 & \text{then } \{\langle F, C\rangle\} \\
 & \cup \\
 & (\text{if } (C.ThisClass \neq \text{Object}) \\
 & \text{then Fields}(C.SuperClass))
\end{array}$$

A.3.9. Cast expression evaluation *type Expression*

At run time, the JVM checks if the concrete type of the value υ of *Expression* evaluation (obtained by calling ConcreteType(υ)) is a subclass of *type*. If this constraint is not satisfied then the exception ClassCastException is thrown from the position of the nearest statement where it exists. If no exception is thrown after the evaluation of such an expression, the value of the expression is returned as a result. This is presented in Table 34.

A.3.10. Field access expression evaluation

We evaluate the *FieldAccess* expression that appear in the righthand side of an assignment expression. So, each expression will return a value. The static fields that are not initialized with constant expressions, at compile time, can trigger the initialization of the class in which they are declared. In this case, before returning the value of the field, the method clinit() is called to initialize this class.
The rules of the field access expression evaluation are presented in the Tables 35, 36, 37 and 38.

A.3.11. Array field access evaluation

An access to an array component of the form *PrimaryNoNewArray*[*Expression*] can cause the NullPointerException if the reference to the array (value of *PrimaryNoNewArray*) is null. When the reference to this array is not null, the value of *Expression* must be a positive integer between 0 and the length of the array. Otherwise, an exception of type IndexOutOfBoundsException will be thrown. The rules of evaluating such an expression are given in the Table 39.

A.3.12. Simple local variable access evaluation

An access to a local variable returns its value from the local variable table of the current frame. The rule is presented in Table 40.

A.3.13. Method call evaluation

For a method invocation, there are many steps that are needed before to jump to the invoked method. First, the value of the receiver is computed. Then the argument is evaluated after which the accessibility to the invoked method is checked (we suppose that the method is accessible). Afterwards, the underlying method code is localized. Finally, a new frame is created to contain the information that is associated with the invoked method.

A.3.14. Computing receiver value

The invocation mode decides what value to give to the receiver. Actually, for a static mode (static $\in$ modifiers of the invoked method) the receiver value is $\perp$ (no receiver) otherwise, it will have some reference value that is the value of this.

A.3.15. Argument evaluation

An argument list is evaluated from the left to the right. In our case, we show how to evaluate just one argument. The same schema could be applied to the case of many arguments.

A.3.16. Method code locali ation

The localization of the invoked method depends on the invocation mode:

- If the invocation mode is static then we know that the invoked method is from the class C (the *ClassFrom* of the method annotation). In this case, the class C can be initialized if it is not already.
- If the invocation mode is private the invoked method is also known but no initialization is triggered.
- Otherwise, a dynamic process is required to retrieve the real method to call. This is achieved in the semantic rules of the method invocation evaluation thanks to the function LookupFirstSuperClass.

We need some functions that allow us to gather the information that is relevant to the invoked method:

- InMethod: a predicate that evaluates to trueif some method exists in some class:

  InMethod:
  $$\mathit{MethodInfo} \times \text{Identifier} \times \mathit{ClassFile} \times \times \mathit{MethodDescriptor} \rightarrow \text{bool}$$
  InMethod:
  $$(M, \text{Identifier}, C, D) = (M \in C.\mathit{Methods}) \wedge (M.\mathit{SimpleName} = \text{Identifier}) \wedge (M.\mathit{Descriptor} = D).$$

- GetInvocMode(M,B): a function that returns the invocation mode of a method invocation expression using the modifier information that is in M. The value of the parameter B is true when the method invocation is *super*.

  GetInvocMode : $\mathit{MethodInfo} \times \text{bool} \rightarrow \text{String}$

  GetInvocMode($M, \mathscr{B}$) =

```
if (static ∈ M.Modifiers)
then   'static'
else   if (private ∈ M.Modifiers)
       then 'nonvirtual'
       else if (𝓑)
            then 'super'
            else if (abstract ∈ M.Modifiers)
                 then 'interface'
                 else 'virtual'
```

- LookupFirstSuperClass(ρ, M, S, I): represents the dynamic process to search in the class hierarchy (explored by Γ) a method M' having the same name and descriptor as M with respect to the invocation mode I. This search is recursive through the class hierarchy and begins from the class S.

```
LookupFirstSuperClass:
address × $MethodInfo$ × $ClassFile$ ×
× String → ($MethodInfo$, $ClassFile$)

LookupFirstSuperClass($\rho, M, S, InvocMode$) =
if (Match($M, M', S$))
then   if (($InvocMode$ =’super’) ∨
       ($InvocMode$ =’interface’))
       then ($M', S$)
       else   if ($InvocMode$ =’virtual’ ∧
                overrides(($M$, con-
                creteType(h($\rho$))), ($M', S$)))
                then ($M$, concreteType($h(\rho)$))
       else   if ($S.ThisClass \neq$ Object)
                then LookupFirstSuperClass($\rho, M$,
                $S.SuperClass, InvocMode$)
else ($\perp, \perp$)
```

- The function overrides verifies if some method overrides another one through the class hierarchy.

```
overrides : ($MethodInfo$,$ClassFile$)×
×($MethodInfo$,$ClassFile$) →bool
overrides(($m$,$C$), ($m'$,$C'$) =
($C \sqsubseteq C'$) ∧
(((private $\notin m'.Modifiers$) ∧ ($C \sqsubseteq C'$))∨
($\exists(m'', C'' \wedge m'' \neq m'' \wedge m'' \neq m')\wedge$
(overrides (($m$,$C$), ($m''$,$C''$))∧
overrides (($m''$,$C''$), ($m'$,$C'$))))) ∧
($m.SimpleName = m'.SimpleName \wedge$
$m.Descriptor = m'.Descriptor$)
```

The underlying rules are presented in Tables 41, 42, 43 and 44.

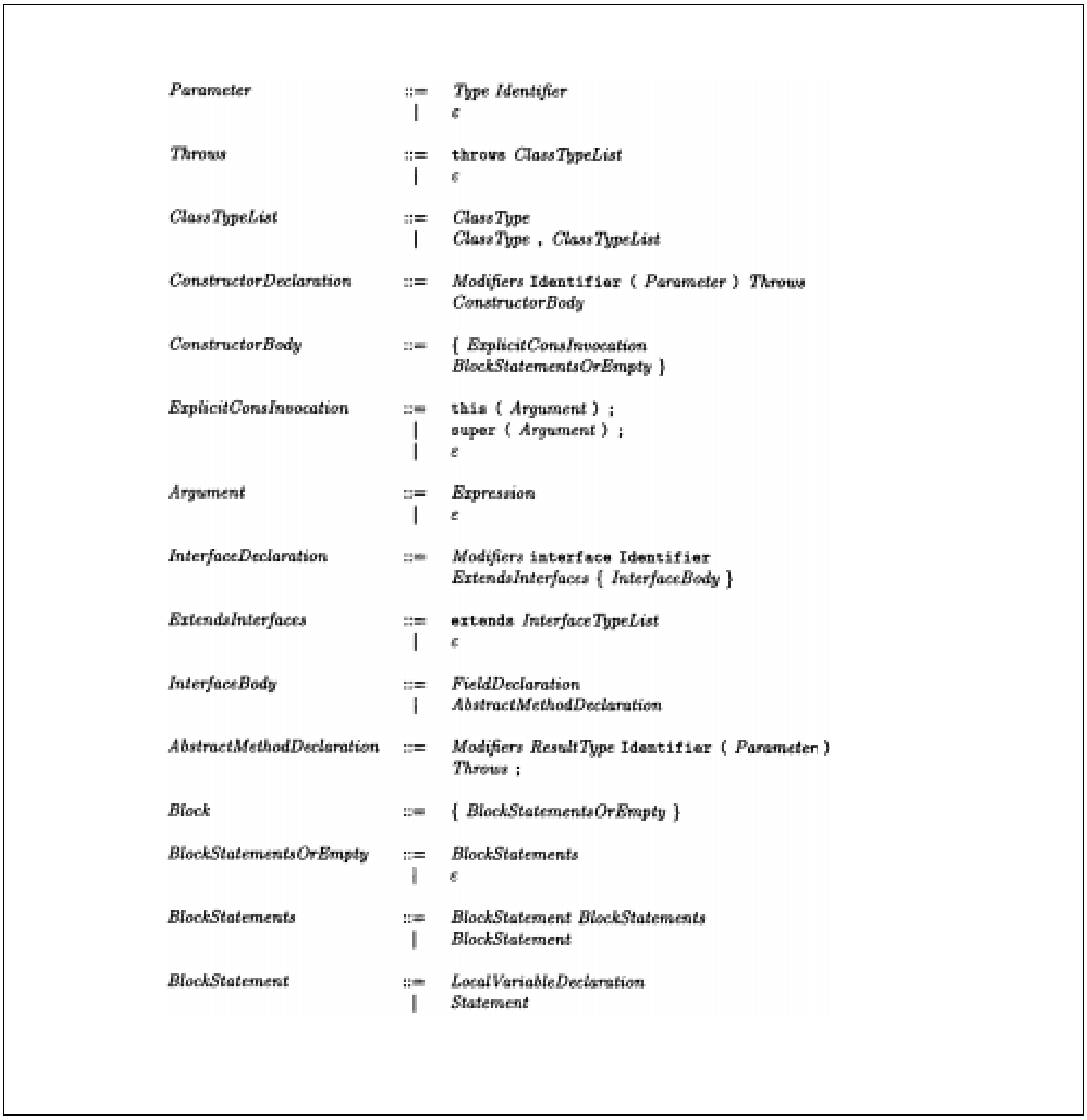
A.3.17. Assignment expression evaluation

An assignment expression is made of a left-hand side, a right-hand side and the operator =. The left-hand side must return a variable, the right-hand side must return a value. We show in the rules of an assignment expression how a field access expression must return a variable. The evaluation result of such an expression is an update of the value of the class or the instance variable with the value of the expression in the righthand side.
Another possible expression in the left hand side is an access to an array component. A runtime check is made between to guarantee the type compliance between the type of the righthand side expression and the type of the left hand expression. If the former is not a subtype of the latter, the arrayStoreException will be thrown at the position of the statement containing this expression. If the left hand side is a *FieldAccess* expression then it returns the variable representing a static or an instance field. An access to a static field can trigger the initialization of the class in which it is declared (if it is not already). The value of the field is updated with the value of the righthand side expression. When the left hand side is an *ArrayAccess* expression, we use a function GetMappeFields(h(ρ)) to return the map *MI* of an array having ρ as address. The rules of the assignment expression evaluation are presented in Tables 45, 46, 47 and 48.

Table 13
Grammar of the subset – Part 1

Program	::=	*ClassDeclaration Program*
		InterfaceDeclaration Program
		ε
ClassDeclaration	::=	*Modifiers* `class Identifier` *Extends Implements*
		{*ClassBody*}
Modifiers	::=	`public` *Modifiers*
		`private` *Modifiers*
		`static` *Modifiers*
		`abstract` *Modifiers*
		`final` *Modifiers*
		`native` *Modifiers*
		`transient` *Modifiers*
		`volatile` *Modifiers*
		ε
Extends	::=	`extends` *ClassType*
		ε
Implements	::=	`implements` *InterfaceTypeList*
		ε
InterfaceTypeList	::=	*InterfaceType*
		InterfaceType , *InterfaceTypeList*
ClassBody	:=	*FieldDeclaration ClassBody*
		MethodDeclaration ClassBody
		AbstractMethodDeclaration ClassBody
		ConstructorDeclaration ClassBody
		ε
FieldDeclaration	::=	*Modifiers Type* `Identifier`
		Modifiers Type `identifier =` *Expression*
	|	*Modifiers SimpleType*`[] identifier =` *ArrayInitiali: er*
ArrayInitialiser	::=	{ }
		{ *ExpressionInitialiser* }
ExpressionInitializer	::=	*Expression*
	|	*Expression* , *ExpressionInitializer*
MethodDeclaration	::=	*Modifiers ResultType* `Identifier` (*Parameter*)
		Throws Block

Table 14
Grammar of the subset – Part 2



Parameter	::=	*Type Identifier*
	|	ε
Throws	::=	`throws` *ClassTypeList*
	|	ε
ClassTypeList	::=	*ClassType*
	|	*ClassType* `,` *ClassTypeList*
ConstructorDeclaration	::=	*Modifiers* `Identifier` `(` *Parameter* `)` *Throws* *ConstructorBody*
ConstructorBody	::=	`{` *ExplicitConsInvocation* *BlockStatementsOrEmpty* `}`
ExplicitConsInvocation	::=	`this` `(` *Argument* `)` `;`
	|	`super` `(` *Argument* `)` `;`
	|	ε
Argument	::=	*Expression*
	|	ε
InterfaceDeclaration	::=	*Modifiers* `interface` `Identifier` *ExtendsInterfaces* `{` *InterfaceBody* `}`
ExtendsInterfaces	::=	`extends` *InterfaceTypeList*
	|	ε
InterfaceBody	::=	*FieldDeclaration*
	|	*AbstractMethodDeclaration*
AbstractMethodDeclaration	::=	*Modifiers* *ResultType* `Identifier` `(` *Parameter* `)` *Throws* `;`
Block	::=	`{` *BlockStatementsOrEmpty* `}`
BlockStatementsOrEmpty	::=	*BlockStatements*
	|	ε
BlockStatements	::=	*BlockStatement* *BlockStatements*
	|	*BlockStatement*
BlockStatement	::=	*LocalVariableDeclaration*
	|	*Statement*

Table 15
Grammar of the subset – Part 3

LocalVariableDeclaration	::=	*Type* Identifier ;
	|	*Type* Identifier = *Expression*
	|	*SimpleType*[] Identifier = *ArrayInitializer* ;
Statement	::=	;
	|	*Block*
	|	*IfStatement*
	|	*WhileStatement*
	|	*ThrowStatement*
	|	*TryStatement*
	|	*ReturnStatement*
	|	*ExpressionStatement*
IfStatement	::=	if (*Expression*) *Statement* else *Statement*
	|	if (*Expression*) *Statement*
WhileStatement	::=	while (*Expression*) *Statement*
ThrowStatement	::=	throw *Expression* ;
TryStatement	::=	*try Catches finally*
	|	*try Catches*
	|	*try finally*
ReturnStatement	::=	return *Expression* ;
	|	return ;
try	::=	try *Block*
finally	::=	finally *Block*
Catches	::=	*Catch*
	|	*Catch Catches*
Catch	::=	catch (*ClassType* Identifier) *Block*
ExpressionStatement	::=	*StatementExpression* ;
StatementExpression	::=	*AssignmentExpression*
	|	*MethodInvocation*
	|	*ClassInstanceCreation*
Primary	::=	*ArrayCreation*
	|	*PrimaryNoNewArray*
ArrayCreation	::=	new *SimpleType* [*Expression*]

Table 16
Grammar of the subset – Part 4

PrimaryNoNewArray	::=	Literal
	|	this
	|	(*Expression*)
	|	*ClassInstanceCreation*
	|	*SimpleFieldAccess*
	|	*ArrayFieldAccess*
	|	*MethodInvocation*
ClassInstanceCreation	::=	new *ClassType* (*Argument*)
SimpleFieldAccess	::=	*Primary* . Identifier
	|	super . Identifier
	|	*FieldName*
FieldName	::=	Identifier
	|	*ClassOrInterfaceType* . Identifier
	|	*ExpressionName* . Identifier
ExpressionName	::=	*FieldName*
	|	*SimpleLocalVarAccess*
SimpleLocalVarAccess	::=	Identifier
ArrayFieldAccess	::=	*PrimaryNoNewArray* [*Expression*]
MethodInvocation	::=	*MethodName* (*Argument*)
	|	*Primary* . Identifier (*Argument*)
	|	super . Identifier (*Argument*)
MethodName	::=	Identifier
	|	*ClassType* . Identifier
	|	*ExpressionName* . Identifier
Expression	::=	*AssignmentExpression*
	|	*CastExpression*
	|	*Primary*
	|	*SimpleLocalVarAccess*
	|	*ArrayLocalVarAccess*
AssignmentExpression	::=	*SimpleFieldAccess* = *Expression*
	|	*ArrayFieldAccess* = *Expression*
	|	Identifier = *Expression*
	|	Identifier[*Expression*] = *Expression*
CastExpression	::=	(*Type*) *Expression*
ArrayLocalVarAccess	::=	Identifier [*Expression*]

Table 17
Method clinit

```
Method clinit() // supposed to be in a class C

if (not (C.Initialized))
  if ((C != Object) and (interface ∉ C.Modifiers))
   C = C.SuperClass ;
    C.clinit() ;
    C.Initialized = true ;
clinitBody
```

Table 18
Method init

```
Method init(Argument)

Class C = this.getClass() ;
if (not (C.Initialized))
  C.clinit() ;
if (C != Object)
  super() ;
initBody
```

Table 19
Instance variables evaluation

$\boxed{\textit{FieldDeclaration}}$ ....

$$\frac{\begin{array}{c}\mathsf{InField}\ (f, \mathtt{Identifier}, \mathcal{F}.\mathit{Class}, \mathit{Type}) \\ F = (f, \mathcal{F}.\mathit{Class}) \\ \mathtt{static} \notin F.\mathit{Field.Modifiers} \quad \rho = \mathcal{F}.\mathtt{this} \\ v = \mathsf{DefaultValue}(\mathit{Type}) \\ C' = \mathsf{ConcreteType}(h(\rho)) \\ MF = \mathsf{GetMappeFields}(h(\rho)) \\ h' = h \dagger [\rho \mapsto (C', MF \dagger [F \mapsto v])]\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{Modifiers}\ \ \mathit{Type}\ \ \mathtt{Identifier}) \rightarrow (\xi, \mathcal{F}, h', F)}$$

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{Expression}) \rightarrow (\xi', \mathcal{F}', h', \mathit{Expression}')}{\Gamma \vdash (\xi, \mathcal{F}, h, F = \mathit{Expression}) \rightarrow (\xi', \mathcal{F}', h', F = \mathit{Expression}')}$$

$$\frac{\begin{array}{c}\mathtt{static} \notin F.\mathit{Field.Modifiers} \quad \rho = \mathcal{F}.\mathtt{this} \\ C' = \mathsf{ConcreteType}(h(\rho)) \\ MF = \mathsf{GetMappeFields}(h(\rho)) \\ h' = h \dagger [\rho \mapsto (C', MF \dagger [F \mapsto v])]\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, F = v) \rightarrow (\xi, \mathcal{F}, h')}$$

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{Expression}_0) \rightarrow (\xi', \mathcal{F}', h', \mathit{Expression}'_0)}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, \{\mathit{Expression}_0, \ldots, \mathit{Expression}_k\}) \rightarrow \\ (\xi', \mathcal{F}', h', \{\mathit{Expression}'_0, \ldots, \mathit{Expression}_k\})\end{array}}$$

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{Expression}_k) \rightarrow (\xi', \mathcal{F}', h', \mathit{Expression}'_k)}{\Gamma \vdash (\xi, \mathcal{F}, h, \{v_0, \ldots, v_{k-1}, \mathit{Expression}_k\}) \rightarrow (\xi', \mathcal{F}', h', \{v_0, \ldots, v_{k-1}, \mathit{Expression}'_k\})}$$

$$\frac{\begin{array}{c}\mathtt{static} \notin F.\mathit{Field.Modifiers} \quad \rho = \mathcal{F}.\mathtt{this} \\ C' = \mathsf{ConcreteType}(h(\rho)) \\ MF = \mathsf{GetMappeFields}(h(\rho)) \\ \rho' = MF(F) \\ T = \mathsf{ConcreteType}(\rho') \\ h' = h[\rho' \mapsto (T, [0 \mapsto v_0, \ldots, k \mapsto v_k])] \\ h'' = h' \dagger [\rho \mapsto (C', MF \dagger [F \mapsto \rho'])]\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, F = \{v_0, \ldots, v_k\}) \rightarrow (\xi, \mathcal{F}, h'')}$$

Table 20
Static variables evaluation

$\boxed{\mathit{FieldDeclaration}}$ ....

$$\frac{\begin{array}{c}\mathsf{InField}\ (f, \mathtt{Identifier}, \mathcal{F}.\mathit{Class}, \mathit{Type})\\ F = (f, \mathcal{F}.\mathit{Class})\\ \mathtt{static} \in F.\mathit{Field}.\mathit{Modifiers}\\ v = \mathsf{DefaultValue}(\mathit{Type})\\ h' = h \dagger [F \mapsto v]\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{Modifiers}\ \ \mathit{Type}\ \ \mathtt{Identifier}) \rightarrow (\xi, \mathcal{F}, h', F)}$$

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{Expression}) \rightarrow (\xi', \mathcal{F}', h', \mathit{Expression}')}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, F = \mathit{Expression}) \rightarrow\\ (\xi', \mathcal{F}', h', F = \mathit{Expression}')\end{array}}$$

$$\frac{\begin{array}{c}\mathtt{static} \in F.\mathit{Field}.\mathit{Modifiers}\\ h' = h \dagger [F \mapsto v]\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, F = v) \rightarrow (\xi, \mathcal{F}, h')}$$

$$\frac{\begin{array}{c}\mathtt{static} \in F.\mathit{Field}.\mathit{Modifiers}\\ \rho = h(F)\\ T = \mathsf{ConcreteType}(\rho)\\ h' = h \dagger [\rho \mapsto (T, [0 \mapsto v_0, \ldots, k \mapsto v_k])]\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, F = \{v_0, \ldots, v_k\}) \rightarrow (\xi, \mathcal{F}, h')}$$

Table 21
Constructor evaluation

$\boxed{\mathit{ExplicitConsInvocation}}$ ....

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{Argument}) \rightarrow (\xi', \mathcal{F}', h', \mathit{Argument}')}{\Gamma \vdash (\xi, \mathcal{F}, h, [D]\mathtt{this}(\mathit{Argument})) \rightarrow (\xi', \mathcal{F}', h', [D]\mathtt{this}(\mathit{Argument}'))}$$

$$\frac{C = \mathsf{ConcreteType}(\mathcal{F}.\mathtt{this})}{\Gamma \vdash (\xi, \mathcal{F}, h, [D]\mathtt{this}(v)) \rightarrow (\xi, \mathcal{F}, h, \mathtt{this}.[C, D]\mathtt{init}(v))}$$

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{Argument}) \rightarrow (\xi', \mathcal{F}', h', \mathit{Argument}')}{\Gamma \vdash (\xi, \mathcal{F}, h, [D]\mathtt{super}(\mathit{Argument})) \rightarrow (\xi', \mathcal{F}', h', [D]\mathtt{super}(\mathit{Argument}'))}$$

$$\frac{C = \mathcal{F}.\mathit{Class}.\mathit{SuperClass}}{\Gamma \vdash (\xi, \mathcal{F}, h, [D]\mathtt{super}(v)) \rightarrow (\xi, \mathcal{F}, h, \mathtt{super}.[C, D]\mathtt{init}(v))}$$

Table 22
Local variable declaration evaluation

$\boxed{\mathit{LocalVariableDeclaration}}$ ....

$$\frac{\begin{array}{c} v = \mathsf{DefaultValue}(\mathit{Type}) \\ \mathcal{LV} = \mathcal{F}.\mathit{Method.LocalVariableTable} \dagger [\mathtt{Identifier} \mapsto v] \end{array}}{\begin{array}{c} \Gamma \vdash (\xi, \mathcal{F}, h, \mathit{Type}\ \ \mathtt{Identifier}) \to \\ (\xi, \mathcal{F}[\mathit{Method.LocalVariableTable} \leftarrow \mathcal{LV}], h, \mathtt{Identifier}) \end{array}}$$

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{Expression}) \to (\xi', \mathcal{F}', h', \mathit{Expression}')}{\Gamma \vdash (\xi, f, h, \mathtt{Identifier} = \mathit{Expression}) \to (\xi', \mathcal{F}', h', \mathtt{Identifier} = \mathit{Expression}')}$$

$$\frac{\mathcal{LV} = \mathcal{F}.\mathit{Method.LocalVariableTable} \dagger [\mathtt{Identifier} \mapsto v]}{\Gamma \vdash (\xi, \mathcal{F}, h, \mathtt{Identifier} = v) \to (\xi, \mathcal{F}[\mathit{Method.LocalVariableTable} \leftarrow \mathcal{LV}], h)}$$

$$\frac{\begin{array}{c} \rho = \mathsf{fresh}(h) \\ T = \mathsf{ConcreteType}(\mathcal{F}.\mathit{Method.LocalVariableTable}(\mathtt{Identifier})) \\ h' = h \dagger [\rho \mapsto (T, [0 \mapsto v_0, \ldots, k \mapsto v_k])] \\ \mathcal{LV} = \mathcal{F}.\mathit{Method.LocalVariableTable} \dagger [\mathtt{Identifier} \mapsto \rho] \end{array}}{\begin{array}{c} \Gamma \vdash (\xi, \mathcal{F}, h, \mathtt{Identifier} = \{v_0, \ldots, v_k\}) \to \\ (\xi, \mathcal{F}[\mathit{Method.LocalVariableTable} \leftarrow \mathcal{LV}], h') \end{array}}$$

Table 23
Statement evaluation – Part 1

$\boxed{\mathit{Block}}$ ....

$$\frac{\square}{\Gamma \vdash (\xi, \mathcal{F}, h, ;) \to (\xi, \mathcal{F}, h)}$$

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{Statement}) \to (\xi', \mathcal{F}', h', \mathit{Statement}')}{\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{Statement}\ ;\ \mathit{Statements}) \to (\xi', \mathcal{F}', h', \mathit{Statement}'\ ;\ \mathit{Statements})}$$

$$\frac{\begin{array}{c} \Gamma \vdash (\xi, \mathcal{F}, h, \mathit{Statement}) \to (\xi^{*}, \mathcal{F}, h') \\ \mathcal{F}.\mathit{ReturnValue} = \bot \end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{Statement}\ ;\ \mathit{Statements}) \to (\xi^{+}, \mathcal{F}, h', \mathit{Statements})}$$

$\boxed{\mathit{IfStatement}}$ ....

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{Expression}) \to (\xi', \mathcal{F}', h', \mathit{Expression}')}{\begin{array}{c} \Gamma \vdash (\xi, \mathcal{F}, h, \mathtt{if}\ (\mathit{Expression})\ \mathit{Statement}) \to \\ (\xi', \mathcal{F}', h', \mathtt{if}\ (\mathit{Expression}')\ \mathit{Statement}) \end{array}}$$

$$\frac{\square}{\Gamma \vdash (\xi, \mathcal{F}, h, \mathtt{if}\ (\mathtt{true})\ \mathit{Statement}) \to (\xi, \mathcal{F}, h, \mathit{Statement})}$$

$$\frac{\square}{\Gamma \vdash (\xi, \mathcal{F}, h, \mathtt{if}\ (\mathtt{false})\ \mathit{Statement}) \to (\xi, \mathcal{F}, h)}$$

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{Expression}) \to (\xi', \mathcal{F}', h', \mathit{Expression}')}{\begin{array}{c} \Gamma \vdash (\xi, \mathcal{F}, h, \mathtt{if}\ (\mathit{Expression})\ \mathit{Statement}_1\ \mathtt{else}\ \mathit{Statement}_2\ ) \to \\ (\xi', \mathcal{F}', h', \mathtt{if}\ (\mathit{Expression}')\ \mathit{Statement}_1\ \mathtt{else}\ \mathit{Statement}_2) \end{array}}$$

$$\frac{\square}{\begin{array}{c} \Gamma \vdash (\xi, \mathcal{F}, h, \mathtt{if}\ (\mathtt{true})\ \mathit{Statement}_1\ \mathtt{else}\ \mathit{Statement}_2\ \ ) \to \\ (\xi, \mathcal{F}, h, \mathit{Statement}_1) \end{array}}$$

$$\frac{\square}{\begin{array}{c} \Gamma \vdash (\xi, \mathcal{F}, h, \mathtt{if}\ (\mathtt{false})\ \mathit{Statement}_1\ \mathtt{else}\ \mathit{Statement}_2) \to \\ (\xi, \mathcal{F}, h, \mathit{Statement}_2) \end{array}}$$

Table 24
Statement evaluation – Part 2

$\boxed{WhileStatement}$ ....

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, Expression) \rightarrow (\xi', \mathcal{F}', h', Expression')}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{while}\ (Expression)\ Statement) \ \rightarrow \\ (\xi', \mathcal{F}', h', (\texttt{if}\ (Expression')\ \texttt{then}\ \ Statement\ );\ \texttt{while}\ (Expression)\ Statement)\end{array}}$$

$$\frac{\Box}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{while (false)}\ Statement) \rightarrow (\xi, \mathcal{F}, h)}$$

Table 25
Return statement evaluation

$\boxed{ReturnStatement}$ ....

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, Expression) \rightarrow (\xi', \mathcal{F}', h', Expression')}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{return}\ Expression) \rightarrow (\xi', \mathcal{F}', h', \texttt{return}\ Expression')}$$

$$\frac{\begin{array}{c}\mathsf{HandlerInTable}(Any, \mathcal{F}.Method.ExceptionTable, H, \mathcal{P}) \\ Statement = H.Target\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{return}\ v, \mathcal{P}) \rightarrow (\mathcal{F}[ReturnValue \leftarrow v], h, Statement)}$$

$$\frac{\neg\mathsf{HandlerInTable}(Any, \mathcal{F}.Method.ExceptionTable, H, \mathcal{P})}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{return}\ v, \mathcal{P}) \rightarrow (\mathcal{F}.PreviousFrame, h, v)}$$

$$\frac{\begin{array}{c}\mathsf{HandlerInTable}(Any, \mathcal{F}.Method.ExceptionTable, H, \mathcal{P}) \\ Statement = H.Target\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{return}, \mathcal{P}) \rightarrow (\mathcal{F}, h, Statement)}$$

$$\frac{\neg\mathsf{HandlerInTable}(Any, \mathcal{F}.Method.ExceptionTable, H, \mathcal{P})}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{return}, \mathcal{P}) \rightarrow (\mathcal{F}.PreviousFrame, h)}$$

Table 26
Exception constructs

`try`
`{`
`1 X.c = b;`
`2 if (b) return;`
`}`
`catch(`E_1 X_1`)` $Block_1$
`...`

`catch(`E_n X_n`)` $Block_n$
`finally` $Block_{n+1}$

Table 27
Exceptions table

From	*To*	*Target*	*Parameter*	*Type*
1	2	$Block_1$	X_1	E_1
1	2	$Block_2$	X_2	E_2
...	...	...	...	...
1	n	$Block_{n+1}$	`void`	*Any*

Table 28

Exception handling – Part 1

$\boxed{\textit{ThrowStatement}}$ ....

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, \textit{Expression}) \rightarrow (\xi', \mathcal{F}', h', \textit{Expression}')}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{throw}\ \ \textit{Expression}) \rightarrow (\xi', \mathcal{F}', h', \texttt{throw}\ \ \textit{Expression}')}$$

$$\frac{\begin{array}{c}\rho = \mathsf{fresh}(h)\\ h' = h \dagger [\rho \mapsto (\texttt{NullPointerException}, \ldots)]\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{throw}\ \ \texttt{null}, \mathcal{P}) \rightarrow (\rho, \mathcal{F}, h', \texttt{throw}\ \ \rho, \mathcal{P})}$$

$$\frac{\begin{array}{c}\mathsf{HandlerInTable}(\mathsf{ConcreteType}(\rho), \mathcal{F}.\textit{Method}.\textit{ExceptionTable}, H, \mathcal{P})\\ \textit{Statement} = H.\textit{Target} \quad X = H.\textit{Parameter}\\ \mathcal{LV} = \mathcal{F}.\textit{Method}.\textit{LocalVariableTable} \dagger [X \mapsto \rho]\end{array}}{\Gamma \vdash (\rho, \mathcal{F}, h, \texttt{throw}\ \ \rho, \mathcal{P}) \rightarrow (\mathcal{F}[\textit{Method}.\textit{LocalVariableTable} \leftarrow \mathcal{LV}], h, \textit{Statement})}$$

$$\frac{\begin{array}{c}\mathsf{HandlerInTable}(\mathsf{ConcreteType}(\rho), \mathcal{F}.\textit{Method}.\textit{ExceptionTable}, H, \mathcal{P})\\ \textit{Statement} = H.\textit{Target} \quad H.\textit{Type} = \textit{Any}\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{throw}\ \ \rho, \mathcal{P}) \rightarrow (\rho, \mathcal{F}, h, \textit{Statement})}$$

$$\frac{\begin{array}{c}\neg\mathsf{HandlerInTable}(\mathsf{ConcreteType}(\rho), \mathcal{F}.\textit{Method}.\textit{ExceptionTable}, H, \mathcal{P})\\ \mathcal{F}.\textit{Method}.\textit{SimpleName} \neq \ \ \texttt{'Main'}\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{throw}\ \ \rho, \mathcal{P}) \rightarrow (\rho, \mathcal{F}.\textit{PreviousFrame}, h, \texttt{throw}\ \ \rho, \mathcal{F}.\textit{ReturnAddress})}$$

$$\frac{\begin{array}{c}\neg\mathsf{HandlerInTable}(\mathsf{ConcreteType}(\rho), \mathcal{F}.\textit{Method}.\textit{ExceptionTable}, H, \mathcal{P})\\ \mathcal{F}.\textit{Method}.\textit{SimpleName} = \ \ \texttt{'Main'}\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{throw}\ \ \rho, \mathcal{P}) \rightarrow (\rho, \mathcal{F}, h, \texttt{System.exit(1)})}$$

$\boxed{\textit{TryStatement}}$ ....

try

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, \textit{Block}) \rightarrow (\xi', \mathcal{F}', h', \textit{Block}')}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{try}\ \ \textit{Block}) \rightarrow (\xi', \mathcal{F}', h', \texttt{try}\ \ \textit{Block}')}$$

Table 29

Exception handling – Part 2

$$\frac{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, Block) \rightarrow (\xi^{+}, \mathcal{F}, h') \\ \mathcal{P} = \mathsf{FinalPosition}(Block) \\ \mathsf{HandlerInTable}(Any, \mathcal{F}.Method.ExceptionTable, H, \mathcal{P}) \\ Statement = H.Target \quad H.Type = Any\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \mathtt{try}\ \ Block) \rightarrow (\xi^{+}, \mathcal{F}, h', Statement)}$$

$$\frac{\begin{array}{c}\rho = \mathsf{fresh}(h) \\ \Gamma \vdash (\xi, \mathcal{F}, h, Block) \rightarrow (\rho, \mathcal{F}, h', \mathtt{throw}\ \ \rho, \mathcal{P})\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \mathtt{try}\ \ Block) \rightarrow (\rho, \mathcal{F}, h', \mathtt{throw}\ \ \rho, \mathcal{P})}$$

catch

$$\frac{\Gamma \vdash (\mathcal{F}, h, Block) \rightarrow (\mathcal{F}', h', Block')}{\Gamma \vdash (\mathcal{F}, h, \mathtt{catch}(ClassType\ \ \mathtt{Identifier})\ \ Block) \rightarrow (\mathcal{F}', h', \mathtt{catch}\ \ Block')}$$

$$\frac{\begin{array}{c}\Gamma \vdash (\mathcal{F}, h, Block) \rightarrow (\mathcal{F}, h') \\ \mathcal{P} = \mathsf{FinalPosition}(Block) \\ \mathsf{HandlerInTable}(Any, \mathcal{F}.Method.ExceptionTable, H, \mathcal{P}) \\ Statement = H.Target \quad H.Type = Any\end{array}}{\Gamma \vdash (\mathcal{F}, h, \mathtt{catch}(ClassType\ \ \mathtt{Identifier})\ \ Block) \rightarrow (\mathcal{F}, h', Statement)}$$

$$\frac{\begin{array}{c}\rho = \mathsf{fresh}(h) \\ \Gamma \vdash (\mathcal{F}, h, Block) \rightarrow (\rho, \mathcal{F}, h', \mathtt{throw}\ \ \rho, \mathcal{P})\end{array}}{\Gamma \vdash (\mathcal{F}, h, \mathtt{catch}(ClassType\ \ \mathtt{Identifier})\ \ Block) \rightarrow (\rho, \mathcal{F}, h', \mathtt{throw}\ \ \rho, \mathcal{P})}$$

finally

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, Block) \rightarrow (\xi', \mathcal{F}', h', Block')}{\Gamma \vdash (\xi, \mathcal{F}, h, \mathtt{finally}\ \ Block) \rightarrow (\xi', \mathcal{F}', h', \mathtt{finally}\ \ Block')}$$

Table 30
Exception handling – Part 3

finally

$$\frac{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, Block) \rightarrow (\xi, \mathcal{F}, h') \\ \mathcal{P} = \mathsf{FinalPosition}(Block) \\ \mathcal{F}.ReturnValue = \bot\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \mathtt{finally}\ \ Block) \rightarrow (\xi, \mathcal{F}, h', \mathtt{throw}\ \ \xi, \mathcal{P})}$$

$$\frac{\begin{array}{c}\rho = \mathsf{fresh}(h) \\ \Gamma \vdash (\xi, \mathcal{F}, h, Block) \rightarrow (\rho, \mathcal{F}, h', \mathtt{throw}\ \ \rho, \mathcal{P})\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \mathtt{finally}\ \ Block) \rightarrow (\rho, \mathcal{F}, h', \mathtt{throw}\ \ \rho, \mathcal{P})}$$

$$\frac{\begin{array}{c}\Gamma \vdash (\mathcal{F}, h, Block) \rightarrow (\mathcal{F}, h') \\ \mathcal{P} = \mathsf{FinalPosition}(Block) \\ \mathcal{F}.ReturnValue \neq \bot \\ \neg\mathsf{HandlerInTable}(Any, \mathcal{F}.Method.ExceptionTable, H, \mathcal{P})\end{array}}{\Gamma \vdash (\mathcal{F}, h, \mathtt{finally}\ \ Block) \rightarrow (\mathcal{F}.PreviousFrame, h', \mathcal{F}.ReturnValue)}$$

$$\frac{\begin{array}{c}\Gamma \vdash (\mathcal{F}, h, Block) \rightarrow (\mathcal{F}, h') \\ \mathcal{P} = \mathsf{FinalPosition}(Block) \\ \mathsf{HandlerInTable}(Any, \mathcal{F}.Method.ExceptionTable, H, \mathcal{P}) \\ Statement = H.Target \quad H.Type = Any\end{array}}{\Gamma \vdash (\mathcal{F}, h, \mathtt{finally}\ \ Block) \rightarrow (\mathcal{F}, h', Statement)}$$

$$\frac{\begin{array}{c}\Gamma \vdash (\mathcal{F}, h, Block) \rightarrow (\mathcal{F}, h') \\ \mathcal{P} = \mathsf{FinalPosition}(Block) \\ \mathcal{F}.ReturnValue = \bot \\ \neg\mathsf{HandlerInTable}(Any, \mathcal{F}.Method.ExceptionTable, H, \mathcal{P})\end{array}}{\Gamma \vdash (\mathcal{F}, h, \mathtt{finally}\ \ Block) \rightarrow (\mathcal{F}, h')}$$

Table 31
New array expression evaluation

$\boxed{ArrayCreation}$ ....

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, Expression) \rightarrow (\xi', \mathcal{F}', h', Expression')}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, \mathtt{new}\ \ SimpleType[Expression]) \ \rightarrow \\ (\xi', \mathcal{F}', h', \mathtt{new}\ \ SimpleType[Expression'])\end{array}}$$

$$\frac{\begin{array}{c}\rho = \mathsf{fresh}(h) \\ v < 0 \\ h' = h \dagger [\rho \mapsto (\mathtt{NegativeArraySizeException}, \ldots)]\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \mathtt{new}\ \ SimpleType[v], \mathcal{P}) \rightarrow (\rho, \mathcal{F}, h', \mathtt{throw}\ \ \rho, \mathcal{P})}$$

$$\frac{\begin{array}{c}\rho = \mathsf{fresh}(h) \\ v \geq 0 \\ v_i = \mathtt{Defaultvalue}(SimpleType) \ \ i \in \{0, \ldots, v-1\} \\ MI = [0 \mapsto v_0, 1 \mapsto v_1, \ldots, v-1 \mapsto v_{v-1}] \\ h' = h \dagger [\rho \mapsto (SimpleType[], MI)]\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \mathtt{new}\ \ SimpleType[v]) \rightarrow (\xi, \mathcal{F}, h', \rho)}$$

Table 32
Literal, this and parenthesized expression evaluation

$\boxed{\mathit{PrimayNoNewArray}}$ ....

$$\frac{v = \mathsf{eval}(\mathit{Literal})}{\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{Literal}) \rightarrow (\xi, \mathcal{F}, h, v)}$$

$$\frac{\rho = \mathcal{F}.\mathtt{this}}{\Gamma \vdash (\xi, \mathcal{F}, h, \mathtt{this}) \rightarrow (\xi, \mathcal{F}, h, \rho)}$$

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{Expression}) \rightarrow (\xi', \mathcal{F}', h', \mathit{Expression}')}{\Gamma \vdash (\xi, \mathcal{F}, h, (\mathit{Expression})) \rightarrow (\xi', \mathcal{F}', h', (\mathit{Expression}'))}$$

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{Expression}) \rightarrow (\xi^{+}, \mathcal{F}, h', v)}{\Gamma \vdash (\xi, \mathcal{F}, h, (\mathit{Expression})) \rightarrow (\xi^{+}, \mathcal{F}, h', v)}$$

Table 33
New instance class creation evaluation

$\boxed{\mathit{ClassInstanceCreation}}$ ....

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{Argument}) \rightarrow (\xi', \mathcal{F}', h', \mathit{Argument}')}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, [D]\mathtt{new}\ \mathit{ClassType}(\mathit{Argument})) \rightarrow \\ (\xi', \mathcal{F}', h', [D]\mathtt{new}\ \mathit{ClassType}(\mathit{Argument}'))\end{array}}$$

$$\frac{\begin{array}{c}\rho = \mathsf{fresh}(h) \\ F_i \in \mathsf{Fields}(\mathit{ClassType}) \quad i \in \{0, \ldots, n-1\} \\ v_i = \mathtt{Defaultvalue}(F_i.\mathit{Field.Descriptor}) \\ h' = h \dagger [\rho \mapsto (\mathit{ClassType}, [F_0 \mapsto v_0, \ldots, F_{n-1} \mapsto v_{n-1}])]\end{array}}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, [D]\mathtt{new}\ \ \mathit{ClassType}(v)) \rightarrow \\ (\xi, \mathcal{F}[\mathtt{this} \leftarrow \rho], h', \mathtt{this}.[\mathit{ClassType}, D]\mathtt{init}(v)\ ;\ \rho)\end{array}}$$

Table 34
Cast expression evaluation

$\boxed{\mathit{CastExpression}}$ ....

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{Expression}) \rightarrow (\xi', \mathcal{F}', h', \mathit{Expression}')}{\Gamma \vdash (\xi, \mathcal{F}, h, (\mathit{Type})\mathit{Expression}) \rightarrow (\xi', \mathcal{F}', h', (\mathit{Type})\mathit{Expression}')}$$

$$\frac{\begin{array}{c}\mathsf{isRefType}(\mathit{Type}) \\ \mathsf{ConcreteType}(h(v)) \sqsubseteq \mathit{Type}\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, (\mathit{Type})v) \rightarrow (\xi, \mathcal{F}, h, v)}$$

$$\frac{\begin{array}{c}\mathsf{isRefType}(\mathit{Type}) \\ \mathsf{ConcreteType}(h(v)) \not\sqsubseteq \mathit{Type} \\ h' = h \dagger [\rho \mapsto (\mathtt{ClassCastException}, \ldots)]\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, (\mathit{Type})v, \mathcal{P}) \rightarrow (\rho, \mathcal{F}, h', \mathtt{throw}\ \ \rho, \mathcal{P})}$$

$$\frac{\neg\mathsf{isRefType}(\mathit{Type})}{\Gamma \vdash (\xi, \mathcal{F}, h, (\mathit{Type})v) \rightarrow (\xi, \mathcal{F}, h, v)}$$

Table 35

Field access expression evaluation – Part 1

$\boxed{\mathit{SimpleFieldAccess}}$

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{Primary}) \rightarrow (\xi', \mathcal{F}', h', \mathit{Primary}')}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{Primary}.[C, D]\texttt{Identifier}) \rightarrow \\ (\xi', \mathcal{F}', h', \mathit{Primary}'.[C, D]\texttt{Identifier})\end{array}}$$

$$\frac{\begin{array}{c}\mathsf{InField}\ (f, \texttt{Identifier}, C, D) \\ F = \langle f, C \rangle \\ \texttt{static} \in F.\mathit{Field.Modifiers} \quad \texttt{final} \in F.\mathit{Field.Modifiers} \\ \mathsf{constant}(F)\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \rho.[C, D]\texttt{Identifier}) \rightarrow (\xi, \mathcal{F}, h, h(F))}$$

$$\frac{\begin{array}{c}\mathsf{InField}\ (f, \texttt{Identifier}, C, D) \\ F = \langle f, C \rangle \\ \texttt{static} \in F.\mathit{Field.Modifiers} \quad \texttt{final} \in F.\mathit{Field.Modifiers} \\ \neg\mathsf{constant}(F) \quad \mathsf{initialized}(C)\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \rho.[C, D]\texttt{Identifier}) \rightarrow (\xi, \mathcal{F}, h, h(F))}$$

$$\frac{\begin{array}{c}\mathsf{InField}\ (f, \texttt{Identifier}, C, D) \\ F = \langle f, C \rangle \\ \texttt{static} \in F.\mathit{Field.Modifiers} \quad \texttt{final} \in F.\mathit{Field.Modifiers} \\ \neg\mathsf{constant}(F) \quad \neg\mathsf{initialized}(C)\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \rho.[C, D]\texttt{Identifier}) \rightarrow (\xi, \mathcal{F}, h, C.[C, ()\texttt{void}]\texttt{clinit()}\ ;\ h(F))}$$

$$\frac{\begin{array}{c}\mathsf{InField}\ (f, \texttt{Identifier}, C, D) \\ F = \langle f, C \rangle \\ \texttt{static} \in F.\mathit{Field.Modifiers} \quad \texttt{final} \notin F.\mathit{Field.Modifiers} \\ \neg\mathsf{initialized}(C)\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \rho.[C, D]\texttt{Identifier}) \rightarrow (\xi, \mathcal{F}, h, C.[C, ()\texttt{void}]\texttt{clinit()}\ ;\ h(F))}$$

Table 36

Field access expression evaluation – Part 2

$\boxed{\mathit{SimpleFieldAccess}}$ ....

$$\frac{\begin{array}{c}\mathsf{InField}\ (f, \mathtt{Identifier}, C, D)\\ F = \langle f, C\rangle\\ \mathtt{static} \in F.\mathit{Field.Modifiers} \quad \mathtt{final} \notin F.\mathit{Field.Modifiers}\\ \mathsf{initialized}(C)\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \rho.[C, D]\mathtt{Identifier}) \rightarrow (\xi, \mathcal{F}, h, h(F))}$$

$$\frac{\begin{array}{c}\mathsf{InField}\ (f, \mathtt{Identifier}, C, D)\\ F = \langle f, C\rangle\\ \rho = \mathsf{fresh}(h)\\ \mathtt{static} \notin F.\mathit{Field.Modifiers}\\ h' = h \dagger [\rho \mapsto (\mathtt{NullPointerException}, \dots)]\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \mathtt{null}.[C, D]\mathtt{Identifier}) \rightarrow (\rho, \mathcal{F}, h, \mathtt{throw}\ \ \rho, \mathcal{P})}$$

$$\frac{\begin{array}{c}\mathsf{InField}\ (f, \mathtt{Identifier}, C, D)\\ F = \langle f, C\rangle\\ \mathtt{static} \notin F.\mathit{Field.Modifiers}\\ MF = \mathsf{GetMappeFields}(h(\rho))\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \rho.[C, D]\mathtt{Identifier}) \rightarrow (\xi, \mathcal{F}, h', MF(F))}$$

$$\frac{\varsigma = \mathcal{F}.\mathit{Class.SuperClass}}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, \mathtt{super}.[C, D]\mathtt{Identifier}) \ \ \rightarrow\\ (\xi, \mathcal{F}, h, ((\varsigma)\mathtt{this}).[C, D]\mathtt{Identifier})\end{array}}$$

$$\frac{\begin{array}{c}\mathsf{InField}\ (f, \mathtt{Identifier}, C, D)\\ F = \langle f, C\rangle\\ \mathtt{static} \in F.\mathit{Field.Modifiers} \quad \mathtt{final} \in F.\mathit{Field.Modifiers}\\ \mathsf{constant}(F)\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{ClassType}.[C, D]\mathtt{Identifier}) \rightarrow (\xi, \mathcal{F}, h, h(F))}$$

Table 37

Field access expression evaluation – Part 3

$\boxed{\mathit{SimpleFieldAccess}}$ ....

$$\frac{\begin{array}{c}\mathsf{InField}\ (f, \mathtt{Identifier}, C, D)\\ F = \langle f, C\rangle\\ \mathtt{static} \in F.\mathit{Field.Modifiers} \quad \mathtt{final} \in F.\mathit{Field.Modifiers}\\ \neg\mathsf{constant}(F) \quad \neg\mathsf{initialized}(C)\end{array}}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{ClassType}.[C, D]\mathtt{Identifier}) \rightarrow\\ (\xi, \mathcal{F}, h, C.[C, ()\mathtt{void}]\mathtt{clinit}()\ ;\ h(F))\end{array}}$$

$$\frac{\begin{array}{c}\mathsf{InField}\ (f, \mathtt{Identifier}, C, D)\\ F = \langle f, C\rangle\\ \mathtt{static} \in F.\mathit{Field.Modifiers} \quad \mathtt{final} \in F.\mathit{Field.Modifiers}\\ \neg\mathsf{constant}(F) \quad \mathsf{initialized}(C)\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{ClassType}.[C, D]\mathtt{Identifier}) \rightarrow (\xi, \mathcal{F}, h, h(F))}$$

$$\frac{\begin{array}{c}\mathsf{InField}\ (f, \mathtt{Identifier}, C, D)\\ F = \langle f, C\rangle\\ \mathtt{static} \in F.\mathit{Field.Modifiers} \quad \mathtt{final} \notin F.\mathit{Field.Modifiers}\\ \neg\mathsf{initialized}(C)\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{ClassType}.[C, D]\mathtt{Identifier}) \rightarrow (\xi, \mathcal{F}, h, C.[C, ()\mathtt{void}]\mathtt{clinit}()\ ;\ h(F))}$$

$$\frac{\begin{array}{c}\mathsf{InField}\ (f, \mathtt{Identifier}, C, D)\\ F = \langle f, C\rangle\\ \mathtt{static} \in F.\mathit{Field.Modifiers} \quad \mathtt{final} \notin F.\mathit{Field.Modifiers}\\ \mathsf{initialized}(C)\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{ClassType}.[C, D]\mathtt{Identifier}) \rightarrow (\xi, \mathcal{F}, h, h(F))}$$

$$\frac{\begin{array}{c}\mathsf{InField}\ (f, \mathtt{Identifier}, C, D)\\ F = \langle f, C\rangle\\ \neg\mathsf{constant}(F) \quad \neg\mathsf{initialized}(C)\end{array}}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{InterfaceType}.[C, D]\mathtt{Identifier}) \rightarrow\\ (\xi, \mathcal{F}, h, C.[C, ()\mathtt{void}]\mathtt{clinit}()\ ;\ h(F))\end{array}}$$

$$\frac{\begin{array}{c}\mathsf{InField}\ (f, \mathtt{Identifier}, C, D)\\ F = \langle f, C\rangle\\ \neg\mathsf{constant}(F) \quad \mathsf{initialized}(C)\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{InterfaceType}.[C, D]\mathtt{Identifier}) \rightarrow (\xi, \mathcal{F}, h, h(F))}$$

Table 38
Field access expression evaluation – Part 4

$\boxed{\mathit{SimpleFieldAccess}}$ ....

$$\frac{\begin{array}{c}\mathsf{InField}\ (f, \mathtt{Identifier}, C, D)\\ F = \langle f, C\rangle\\ \mathsf{constant}(F)\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{InterfaceType}.[C,D]\mathtt{Identifier}) \rightarrow (\xi, \mathcal{F}, h, h(F))}$$

$$\frac{\begin{array}{c}\mathsf{InField}\ (f, \mathtt{Identifier}, C, D)\\ F = \langle f, C\rangle\\ \mathtt{static} \notin F.\mathit{Field}.\mathit{Modifiers}\\ \mathit{MF} = \mathsf{GetMappeFields}(h(\rho))\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, [C,D]\mathtt{Identifier}) \rightarrow (\xi, \mathcal{F}, h, \mathit{MF}(F))}$$

$$\frac{\begin{array}{c}\mathsf{InField}\ (f, \mathtt{Identifier}, C, D)\\ F = \langle f, C\rangle\\ \mathtt{static} \in F.\mathit{Field}.\mathit{Modifiers}\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, [C,D]\mathtt{Identifier}) \rightarrow (\xi, \mathcal{F}, h, h(F))}$$

Table 39
Array field access evaluation

$\boxed{\mathit{ArrayFieldAccess}}$ ....

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{PrimaryNoNewArray}) \rightarrow (\xi', \mathcal{F}', h', \mathit{PrimaryNoNewArray}')}{\begin{array}{c}\Gamma \vdash (\mathcal{F}, h, \mathit{PrimaryNoNewArray}[\mathit{Expression}]) \rightarrow\\ (\xi', \mathcal{F}', h', \mathit{PrimaryNoNewArray}'[\mathit{Expression}])\end{array}}$$

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{Expression}) \rightarrow (\xi', \mathcal{F}', h', \mathit{Expression}')}{\Gamma \vdash (\xi, \mathcal{F}, h, \rho[\mathit{Expression}]) \rightarrow (\xi', \mathcal{F}', h', \rho[\mathit{Expression}'])}$$

$$\frac{\begin{array}{c}\rho = \mathsf{fresh}(h)\\ h' = h \dagger [\rho \mapsto (\mathtt{NullPointerException}, \ldots)]\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \mathtt{null}[v], \mathcal{P}) \rightarrow (\rho, \mathcal{F}, h', \mathtt{throw}\ \ \rho, \mathcal{P})}$$

$$\frac{\begin{array}{c}v\ <\ 0\ \ \vee\ \ v\ \geq\ h(\rho).\mathtt{length}\\ \rho' = \mathsf{fresh}(h)\\ h' = h \dagger [\rho' \mapsto (\mathtt{IndexOutOfBoundsException}, \ldots)\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \rho[v], \mathcal{P}) \rightarrow (\rho', \mathcal{F}, h', \mathtt{throw}\ \ \rho', \mathcal{P})}$$

$$\frac{\begin{array}{c}v\ \geq\ 0\ \ \wedge\ \ v\ <\ h(\rho).\mathtt{length}\\ \mathit{MI} = \mathsf{GetMappeIndex}(h(\rho))\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \rho[v]) \rightarrow (\xi, \mathcal{F}, h, \mathit{MI}(v))}$$

Table 40
Simple local variable access

$\boxed{\mathit{SimpleLocalVariable}}$ ....

$$\frac{v = \mathcal{F}.\mathit{Method}.\mathit{LocalVariableTable}(\mathtt{Identifier})}{\Gamma \vdash (\xi, \mathcal{F}, h, \mathtt{Identifier}) \rightarrow (\xi, \mathcal{F}, h, v)}$$

Table 41
Method call evaluation – Part 1

$\boxed{MethodInvocation}$ ....

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, Argument) \rightarrow (\xi', \mathcal{F}', h', Argument')}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, [C, D]\texttt{Identifier}(Argument)) \rightarrow \\ (\xi', \mathcal{F}', h', [C, D]\texttt{Identifier}(Argument'))\end{array}}$$

$$\frac{\begin{array}{c}\mathsf{InMethod}\ (M, \texttt{Identifier}, C, D) \\ \texttt{static} \in M.Modifiers \ \vee \ \texttt{private} \in M.Modifiers \\ \texttt{Id} = \mathsf{GetFormalParameter}(M) \\ \mathcal{F}' = \langle \bot, C, M, \mathcal{F}, \mathcal{P}, \bot \rangle \\ \mathcal{LV} = \mathcal{F}'.Method.LocalVariableTable \dagger [\texttt{Id} \mapsto v]\end{array}}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, [C, D]\texttt{Identifier}(v), \mathcal{P}) \rightarrow \\ (\xi, \mathcal{F}'[Method.LocalVariableTable \leftarrow \mathcal{LV}], h, \mathcal{F}'.Method.Code)\end{array}}$$

$$\frac{\begin{array}{c}\mathsf{InMethod}\ (M, \texttt{Identifier}, C, D) \\ \texttt{static} \notin M.Modifiers \quad \texttt{private} \notin M.Modifiers \\ InvocMode = \mathsf{GetInvocMode}(M, \texttt{false}) \\ S = \mathsf{ConcreteType}(\mathcal{F}.\texttt{this}) \\ (M', R) = \mathsf{LookupFirstSuperClass}(\mathcal{F}.this, M, S, InvocMode) \\ \texttt{Id} = \mathsf{GetFormalParameter}(M') \\ \mathcal{F}' = \langle \mathcal{F}.\texttt{this}, R, M', \mathcal{F}, \mathcal{P}, \bot \rangle \\ \mathcal{LV} = \mathcal{F}'.Method.LocalVariableTable \dagger [\texttt{Id} \mapsto v]\end{array}}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, [C, D]\texttt{Identifier}(v), \mathcal{P}) \rightarrow \\ (\xi, \mathcal{F}'[Method.LocalVariableTable \leftarrow \mathcal{LV}], h, \mathcal{F}'.Method.Code)\end{array}}$$

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, Argument) \rightarrow (\xi', \mathcal{F}', h', Argument')}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, ClassType.[C, D]\texttt{Identifier}(Argument)) \rightarrow \\ (\xi', \mathcal{F}', h', ClassType.[C, D]\texttt{Identifier}(Argument'))\end{array}}$$

$$\frac{\begin{array}{c}\mathsf{InMethod}\ (M, \texttt{Identifier}, C, D) \\ \texttt{static} \in M.Modifiers \\ \texttt{Id} = \mathsf{GetFormalParameter}(M) \\ \mathcal{F}' = \langle \bot, ClassType, M, \mathcal{F}, \mathcal{P}, \bot \rangle \\ \mathcal{LV} = \mathcal{F}'.Method.LocalVariableTable \dagger [\texttt{Id} \mapsto v] \\ \neg\mathsf{initialized}(C)\end{array}}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, ClassType.[C, D]\texttt{Identifier}(v), \mathcal{P}) \rightarrow \\ (\xi, \mathcal{F}'[Method.LocalVariableTable \leftarrow \mathcal{LV}], h, C.[C, ()\texttt{void}]\texttt{clinit}()\ ; \\ \mathcal{F}'.Method.Code)\end{array}}$$

Table 42
Method call evaluation – Part 2

$\boxed{\mathit{MethodInvocation}}$ ....

$$\frac{\begin{array}{c}\mathsf{InMethod}\ (M, \mathtt{Identifier}, C, D)\\ \mathtt{static} \in M.\mathit{Modifiers}\\ \mathtt{Id} = \mathsf{GetFormalParameter}(M)\\ \mathcal{F}' = \langle \bot, \mathit{ClassType}, M, \mathcal{F}, \mathcal{P}, \bot\rangle\\ \mathcal{LV} = \mathcal{F}'.\mathit{Method.LocalVariableTable} \dagger [\mathtt{Id} \mapsto v]\\ \mathsf{initialized}(C)\end{array}}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{ClassType}.[C, D]\mathtt{Identifier}(v), \mathcal{P}) \ \rightarrow\\ (\xi, \mathcal{F}'[\mathit{Method.LocalVariableTable} \leftarrow \mathcal{LV}], h, \mathcal{F}'.\mathit{Method.Code})\end{array}}$$

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{Primary}) \rightarrow (\xi', \mathcal{F}', h', \mathit{Primary}')}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{Primary}.[C, D]\mathtt{Identifier}(\mathit{Argument})) \rightarrow\\ (\xi', \mathcal{F}', h', \mathit{Primary}'.[C, D]\mathtt{Identifier}(\mathit{Argument}))\end{array}}$$

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{Argument}) \rightarrow (\xi', \mathcal{F}', h', \mathit{Argument}')}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, \rho.[C, D]\mathtt{Identifier}(\mathit{Argument})) \rightarrow\\ (\xi', \mathcal{F}', h', \rho.[C, D]\mathtt{Identifier}(\mathit{Argument}'))\end{array}}$$

$$\frac{\begin{array}{c}\mathsf{InMethod}\ (M, \mathtt{Identifier}, C, D)\\ \mathtt{static} \in M.\mathit{Modifiers} \quad \mathtt{private} \notin M.\mathit{Modifiers}\\ \mathtt{Id} = \mathsf{GetFormalParameter}(M)\\ \mathcal{F}' = \langle \bot, C, M, \mathcal{F}, \mathcal{P}, \bot\rangle\\ \mathcal{LV} = \mathcal{F}'.\mathit{Method.LocalVariableTable} \dagger [\mathtt{Id} \mapsto v]\\ \neg\mathsf{initialized}(C)\end{array}}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, \rho.[C, D]\mathtt{Identifier}(v), \mathcal{P}) \ \rightarrow\\ (\xi, \mathcal{F}'[\mathit{Method.LocalVariableTable} \leftarrow \mathcal{LV}], h, C.[C, ()\mathtt{void}]\mathtt{clinit}()\ ;\\ \mathcal{F}'.\mathit{Method.Code})\end{array}}$$

$$\frac{\begin{array}{c}\mathsf{InMethod}\ (M, \mathtt{Identifier}, C, D)\\ \mathtt{static} \in M.\mathit{Modifiers} \quad \mathtt{private} \notin M.\mathit{Modifiers}\\ \mathtt{Id} = \mathsf{GetFormalParameter}(M)\\ \mathcal{F}' = \langle \bot, C, M, \mathcal{F}, \mathcal{P}, \bot\rangle\\ \mathcal{LV} = \mathcal{F}'.\mathit{Method.LocalVariableTable} \dagger [\mathtt{Id} \mapsto v]\\ \mathsf{initialized}(C)\end{array}}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, \rho.[C, D]\mathtt{Identifier}(v), \mathcal{P}) \ \rightarrow\\ (\xi, \mathcal{F}'[\mathit{Method.LocalVariableTable} \leftarrow \mathcal{LV}], h, \mathcal{F}'.\mathit{Method.Code})\end{array}}$$

Table 43
Method call evaluation – Part 3

$\boxed{\textit{MethodInvocation}}$ ....

$$\frac{\begin{array}{c}\mathsf{InMethod}\ (M, \mathtt{Identifier}, C, D)\\ \mathtt{private} \in M.\mathit{Modifiers}\\ \mathtt{Id} = \mathsf{GetFormalParameter}(M)\\ \mathcal{F}' = (\bot, C, M, \mathcal{F}, \mathcal{P}, \bot)\\ \mathcal{LV} = \mathcal{F}'.\mathit{Method.LocalVariableTable} \dagger [\mathtt{Id} \mapsto v]\end{array}}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, \rho.[C, D]\mathtt{Identifier}(v), \mathcal{P}) \ \rightarrow\\ (\xi, \mathcal{F}'[\mathit{Method.LocalVariableTable} \leftarrow \mathcal{LV}], h, \mathcal{F}'.\mathit{Method.Code})\end{array}}$$

$$\frac{\begin{array}{c}\mathsf{InMethod}\ (M, \mathtt{Identifier}, C, D)\\ \mathtt{static} \notin M.\mathit{Modifiers}\\ \rho = \mathsf{fresh}(h)\\ h' = h \dagger [\rho \mapsto (\mathtt{NullPointerException}, \ldots)]\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \mathtt{null}.[C, D]\mathtt{Identifier}(v), \mathcal{P}) \rightarrow (\rho, \mathcal{F}, h', \mathtt{throw}\ \ \rho, \mathcal{P})}$$

$$\frac{\begin{array}{c}\mathsf{InMethod}\ (M, \mathtt{Identifier}, C, D)\\ \mathtt{static} \notin M.\mathit{Modifiers} \quad \mathtt{private} \notin M.\mathit{Modifiers}\\ S = \mathsf{ConcreteType}(h(\rho))\\ \mathit{InvocMode} = \mathsf{GetInvocMode}(M, \mathtt{false})\\ (M', R) = \mathsf{LookupFirstSuperClass}(\rho, M, S, \mathit{InvocMode})\\ \mathtt{Id} = \mathsf{GetFormalParameter}(M')\\ \mathcal{F}' = (\rho, R, M', \mathcal{F}, \mathcal{P}, \bot)\\ \mathcal{LV} = \mathcal{F}'.\mathit{Method.LocalVariableTable} \dagger [\mathtt{Id} \mapsto v]\end{array}}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, \rho.[C, D]\mathtt{Identifier}(v), \mathcal{P}) \ \rightarrow\\ (\xi, \mathcal{F}'[\mathit{Method.LocalVariableTable} \leftarrow \mathcal{LV}], h, \mathcal{F}'.\mathit{Method.Code})\end{array}}$$

$$\frac{\begin{array}{c}\mathsf{InMethod}\ (M, \mathtt{Identifier}, C, D)\\ \mathtt{static} \notin M.\mathit{Modifiers} \quad \mathtt{private} \notin M.\mathit{Modifiers}\\ \mathit{InvocMode} = \mathsf{GetInvocMode}(M, \mathtt{false})\\ (\bot, \bot) = \mathsf{LookupFirstSuperClass}(\rho, M, S, \mathit{InvocMode})\end{array}}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, \rho.[C, D]\mathtt{Identifier}(v), \mathcal{P}) \ \rightarrow\\ (\xi, \mathcal{F}, h, \mathtt{throw\ \ new\ \ AbstractMethodError()}, \mathcal{P})\end{array}}$$

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{FieldName}) \rightarrow (\xi', \mathcal{F}', h', \mathit{FieldName}')}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{FieldName}.[C, D]\mathtt{Identifier}(\mathit{Argument})) \ \rightarrow\\ (\xi', \mathcal{F}', h', \mathit{FieldName}'.[C, D]\mathtt{Identifier}(\mathit{Argument}))\end{array}}$$

Table 44
Method call evaluation – Part 4

$\boxed{\mathit{MethodInvocation}}$ ....

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{Argument}) \rightarrow (\xi', \mathcal{F}', h', \mathit{Argument}')}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, \rho.[C, D]\mathtt{Identifier}(\mathit{Argument})) \rightarrow \\ (\xi', \mathcal{F}', h', \rho.[C, D]\mathtt{Identifier}(\mathit{Argument}'))\end{array}}$$

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{Argument}) \rightarrow (\xi', \mathcal{F}', h', \mathit{Argument}')}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, \mathtt{super}.[C, D]\mathtt{Identifier}(\mathit{Argument})) \rightarrow \\ (\xi', \mathcal{F}', h', \mathtt{super}.[C, D]\mathtt{Identifier}(\mathit{Argument}'))\end{array}}$$

$$\frac{\begin{array}{c}\mathsf{InMethod}\ (M, \mathtt{Identifier}, C, D) \\ \mathtt{static} \in M.\mathit{Modifiers} \\ \mathtt{Id} = \mathsf{GetFormalParameter}(M) \\ \mathcal{F}' = (\bot, C, M, \mathcal{F}, \mathcal{P}, \bot) \\ \mathcal{LV} = \mathcal{F}'.\mathit{Method}.\mathit{LocalVariableTable} \dagger [\mathtt{Id} \mapsto v]\end{array}}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, \mathtt{super}.[C, D]\mathtt{Identifier}(v)) \ \rightarrow \\ (\xi, \mathcal{F}'[\mathit{Method}.\mathit{LocalVariableTable} \leftarrow \mathcal{LV}], h, \mathcal{F}'.\mathit{Method}.\mathit{Code})\end{array}}$$

$$\frac{\begin{array}{c}\mathsf{InMethod}\ (M, \mathtt{Identifier}, C, D) \\ \mathtt{static} \notin M.\mathit{Modifiers} \\ \mathit{InvocMode} = \mathsf{GetInvocMode}(M, \mathtt{true}) \\ (M', R) = \mathsf{LookupFirstSuperClass}(\mathcal{F}.\mathtt{this}, M, \mathcal{F}.\mathit{Class}.\mathit{SuperClass}, \mathit{InvocMode}) \\ \mathtt{Id} = \mathsf{GetFormalParameter}(M') \\ \mathcal{F}' = (\mathcal{F}.\mathtt{this}, R, M', \mathcal{F}, \mathcal{P}, \bot) \\ \mathcal{LV} = \mathcal{F}'.\mathit{Method}.\mathit{LocalVariableTable} \dagger [\mathtt{Id} \mapsto v]\end{array}}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, \mathtt{super}.[C, D]\mathtt{Identifier}(v)) \ \rightarrow \\ (\xi, \mathcal{F}'[\mathit{Method}.\mathit{LocalVariableTable} \leftarrow \mathcal{LV}], h, \mathcal{F}'.\mathit{Method}.\mathit{Code})\end{array}}$$

$$\frac{\begin{array}{c}\mathsf{InMethod}\ (M, \mathtt{Identifier}, C, D) \\ \mathtt{static} \notin M.\mathit{Modifiers} \\ \mathit{InvocMode} = \mathsf{GetInvocMode}(M, \mathtt{true}) \\ (\bot, \bot) = \mathsf{LookupFirstSuperClass}(\mathcal{F}.\mathtt{this}, M, \mathcal{F}.\mathit{Class}.\mathit{SuperClass}, \mathit{InvocMode})\end{array}}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, \mathtt{super}.[C, D]\mathtt{Identifier}(v), \mathcal{P}) \ \rightarrow \\ (\xi, \mathcal{F}, h, \mathtt{throw\ new\ AbstractMethodError}(), \mathcal{P})\end{array}}$$

Table 45
Assignment evaluation – Part 1

$\boxed{\mathit{AssignmentExpression}}$

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{SimpleFieldAccess}) \to (\xi', \mathcal{F}', h', \mathit{SimpleFieldAccess}')}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{SimpleFieldAccess} = \mathit{Expression}) \to \\ (\xi', \mathcal{F}', h', \mathit{SimpleFieldAccess}' = \mathit{Expression})\end{array}}$$

$$\frac{(\xi, \mathcal{F}, h, \mathit{Expression}) \to (\xi', \mathcal{F}', h', \mathit{Expression}')}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, \rho.[C, D]\mathtt{Identifier} = \mathit{Expression}) \to \\ (\xi', \mathcal{F}', h', \rho.[C, D]\mathtt{Identifier} = \mathit{Expression}')\end{array}}$$

$$\frac{\begin{array}{c}\mathsf{InField}\ (f, \mathtt{Identifier}, C, D) \\ F = \langle f, C\rangle \\ \mathtt{static} \notin F.\mathit{Field}.\mathit{Modifiers} \quad \mathtt{final} \notin F.\mathit{Field}.\mathit{Modifiers} \\ C' = \mathsf{ConcreteType}(h(\rho)) \\ MF = \mathsf{GetMappeFields}(h(\rho)) \\ h' = h \dagger [\rho \mapsto (C', MF \dagger [F \mapsto v])]\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \rho.[C, D]\mathtt{Identifier} = v) \to (\xi, \mathcal{F}, h', v)}$$

$$\frac{\begin{array}{c}\mathsf{InField}\ (f, \mathtt{Identifier}, C, D) \\ F = \langle f, C\rangle \\ \mathtt{static} \in F.\mathit{Field}.\mathit{Modifiers} \quad \mathtt{final} \notin F.\mathit{Field}.\mathit{Modifiers} \\ \neg\mathsf{initialized}(C)\end{array}}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, \rho.[C, D]\mathtt{Identifier} = v) \to \\ (\xi, \mathcal{F}, h, C.[C, ()\mathtt{void}]\mathtt{clinit}()\ ;\ \rho.[C, D]\mathtt{Identifier} = v)\end{array}}$$

Table 46
Assignment evaluation – Part 2

$\boxed{\mathit{AssignmentExpression}}$

$$\frac{\begin{array}{c}\mathsf{InField}\ (f, \mathtt{Identifier}, C, D) \\ F = \langle f, C\rangle \\ \mathtt{static} \in F.\mathit{Field}.\mathit{Modifiers} \quad \mathtt{final} \notin F.\mathit{Field}.\mathit{Modifiers} \\ \mathsf{initialized}(C) \\ h' = h \dagger [F \mapsto v]\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \rho.[C, D]\mathtt{Identifier} = v) \to (\xi, \mathcal{F}, h', v)}$$

$$\frac{\begin{array}{c}\mathsf{InField}\ (f, \mathtt{Identifier}, C, D) \\ F = \langle f, C\rangle \\ \mathtt{static} \in F.\mathit{Field}.\mathit{Modifiers} \quad \mathtt{final} \notin F.\mathit{Field}.\mathit{Modifiers} \\ \neg\mathsf{initialized}(C)\end{array}}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{ClassType}.[C, D]\mathtt{Identifier} = v) \to \\ (\xi, \mathcal{F}, h, C.[C, ()\mathtt{void}]\mathtt{clinit}()\ ;\ \mathit{ClassType}.[C, D]\mathtt{Identifier} = v)\end{array}}$$

$$\frac{\begin{array}{c}\mathsf{InField}\ (f, \mathtt{Identifier}, C, D) \\ F = \langle f, C\rangle \\ \mathtt{static} \in F.\mathit{Field}.\mathit{Modifiers} \quad \mathtt{final} \notin F.\mathit{Field}.\mathit{Modifiers} \\ \mathsf{initialized}(C) \\ h' = h \dagger [F \mapsto v]\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{ClassType}.[C, D]\mathtt{Identifier} = v) \to (\xi, \mathcal{F}, h', v)}$$

$$\frac{\begin{array}{c}\mathsf{InField}\ (f, \mathtt{Identifier}, C, D) \\ F = \langle f, C\rangle \\ \mathtt{static} \in F.\mathit{Field}.\mathit{Modifiers} \quad \mathtt{final} \notin F.\mathit{Field}.\mathit{Modifiers} \\ h' = h \dagger [F \mapsto v]\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, [C, D]\mathtt{Identifier} = v) \to (\xi, \mathcal{F}, h', v)}$$

Table 47
Assignment evaluation – Part 3

$\boxed{\textit{AssignmentExpression}}$ ....

$$\frac{\begin{array}{c}\mathsf{InField}\ (f, \texttt{Identifier}, C, D)\\ F = \langle f, C\rangle \\ \texttt{static} \notin F.\mathit{Field}.\mathit{Modifiers} \\ \rho = \mathcal{F}.\texttt{this} \\ C' = \mathsf{ConcreteType}(h(\rho)) \\ MF = \mathsf{GetMappeFields}(h(\rho)) \\ h' = h \dagger [\rho \mapsto (C', MF \dagger [F \mapsto v])]\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, [C, D]\texttt{Identifier} = v) \to (\xi, \mathcal{F}, h', v)}$$

$$\frac{\begin{array}{c}\mathsf{InField}\ (f, \texttt{Identifier}, C, D)\\ F = \langle f, C\rangle \\ \texttt{static} \in F.\mathit{Field}.\mathit{Modifiers} \\ h' = h \dagger [F \mapsto v]\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{super}.[C, D]\texttt{Identifier} = v) \to (\xi, \mathcal{F}, h', v)}$$

$$\frac{\begin{array}{c}\mathsf{InField}\ (f, \texttt{Identifier}, C, D)\\ F = \langle f, C\rangle \\ \texttt{static} \notin F.\mathit{Field}.\mathit{Modifiers} \\ \rho = \mathcal{F}.\texttt{this} \\ C' = \mathsf{ConcreteType}(h(\rho)) \\ MF = \mathsf{GetMappeFields}(h(\rho)) \\ h' = h \dagger [\rho \mapsto (C', MF \dagger [F \mapsto v])]\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{super}.[C, D]\texttt{Identifier} = v) \to (\xi, \mathcal{F}, h', v)}$$

$$\frac{(\xi, \mathcal{F}, h, \mathit{Expression}) \to (\xi', \mathcal{F}', h', \mathit{Expression}')}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{Identifier} = \mathit{Expression}) \to (\xi', \mathcal{F}', h', \texttt{Identifier} = \mathit{Expression}')}$$

Table 48
Assignment evaluation – Part 4

$\boxed{\mathit{AssignmentExpression}}$ ....

$$\frac{\mathcal{LV} = \mathcal{F}.\mathit{Method}.\mathit{LocalVariableTable} \dagger [\texttt{Identifier} \mapsto v]}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{Identifier} = v) \to (\xi, \mathcal{F}[\mathit{Method}.\mathit{LocalVariableTable} \leftarrow \mathcal{LV}], h, v)}$$

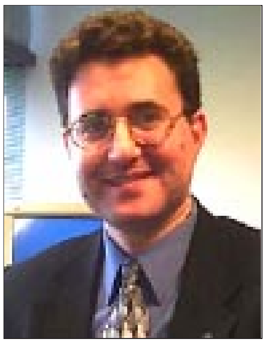
$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{ArrayFieldAccess}) \to (\xi', \mathcal{F}', h', \mathit{ArrayFieldAccess}')}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{ArrayFieldAccess} = \mathit{Expression}) \to \\ (\xi', \mathcal{F}', h', \mathit{ArrayFieldAccess}' = \mathit{Expression})\end{array}}$$

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{Expression}) \to (\xi', \mathcal{F}', h', \mathit{Expression}')}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, \rho[v] = \mathit{Expression}) \to \\ (\xi', \mathcal{F}', h', \rho[v] = \mathit{Expression}')\end{array}}$$

$$\frac{\begin{array}{c} v_1 \ \geq \ 0 \ \wedge \ v_1 \ < \ h(\rho).\texttt{length} \\ MI = \mathsf{GetMappeIndex}(h(\rho)) \\ \mathsf{isRefType}(\mathsf{ConcreteType}(v_2)) \\ (\mathsf{ConcreteType}(v_2) \ \sqsubseteq \ \mathsf{ConcreteType}(h(MI(v_1)))) \\ C' = \mathsf{ConcreteType}(h(\rho)) \\ h' = h \dagger [\rho \mapsto (C', MI \dagger [v_1 \mapsto v_2])] \end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \rho[v_1] = v_2) \to (\xi, \mathcal{F}, h', v_2)}$$

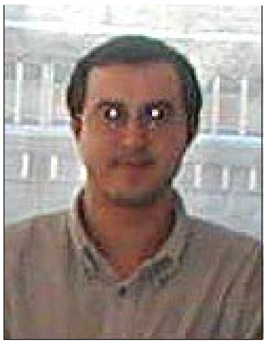
$$\frac{\begin{array}{c} \rho' = \mathsf{fresh}(h) \\ h' = h \dagger [\rho' \mapsto (\texttt{ArrayStoreException}, \ldots)] \\ MI = \mathsf{GetMappeIndex}(h(\rho)) \\ \mathsf{isRefType}(\mathsf{ConcreteType}(v_2)) \\ \neg(\mathsf{ConcreteType}(v_2) \sqsubseteq \mathsf{ConcreteType}(h(MI(v_1)))) \end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \rho[v_1] = v_2, \mathcal{P}) \to (\rho', \mathcal{F}, h', \texttt{throw} \ \ \rho', \mathcal{P})}$$

$$\frac{\begin{array}{c} v_1 \ \geq \ 0 \ \wedge \ v_1 \ < \ h(\rho).\texttt{length} \\ MI = \mathsf{GetMappeIndex}(h(\rho)) \\ \neg\mathsf{isRefType}(\mathsf{ConcreteType}(v_2)) \\ C' = \mathsf{ConcreteType}(h(\rho)) \\ h' = h \dagger [\rho \mapsto (C', MI \dagger [v_1 \mapsto v_2])] \end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \rho[v_1] = v_2) \to (\xi, \mathcal{F}, h', v_2)}$$



Mourad Debbabi holds Ph.D. and M.Sc. degrees in computer science from Paris-XI Orsay, University, France. He is a Lead Scientist at Panasonic Information and Networking Technologies Laboratory, Princeton, New Jersey, USA. He is pursuing research on middleware for next-generation networks. He is the Specification Lead of four JAIN (Java Intelligent Networks) Java Specification Requests (JSRs) dedicated to the elaboration of standard specifications for presence and instant messaging through the Java Community Process (JCP) program. He is also a member of the JAIN Council and participates at the JCP Executive Committee. He is also a Tenured Associate Professor (on leave) at the Computer Science Department of Laval University, Quebec, Canada. In the past, he served as a Senior Scientist at General Electric Research Center, New York, USA; Research Associate at the Computer Science Department of Stanford University, California, USA; and Permanent Researcher at the Bull Corporate Research Center, Paris, France. He published more than 40 research papers in international journals and conferences on Java technology security and acceleration, crypto-protocol analysis, malicious code detection, programming languages, formal semantics, type theory and specification and verification of safety-critical systems.
e-mail: debbabim@research.panasonic.com
Panasonic Information
Networking Technologies Laboratory
Two Research Way
Princeton, New Jersey 08540, USA
e-mail: debbabi@ift.ulaval.ca
LSFM Research Group, Computer Science Department
Laval University
Quebec, G1K 7P4, Canada

Nadia Tawbi holds Ph.D. and M.Sc. degrees in computer science from Paris-VI University, France. She is a Tenured Associate Professor at the Computer Science Department of Laval University, Quebec, Canada. Before that she served as a Group Leader and a Permanent Researcher at the Bull Corporate Research Center, Paris, France. She published several research papers in international journals and conferences on computer security, crypto-protocol analysis, malicious code detection, programming languages, static analysis, formal semantics, and specification and verification of safety-critical systems.
e-mail: tawbi@ift.ulaval.ca
LSFM Research Group, Computer Science Department
Laval University
Quebec, G1K 7P4, Canada



Hamdi ahyaoui is a researcher at the LSFM (Languages Semantics and Formal Methods) Research Group at the Computer Science Department of Laval University, Quebec, Canada. He is pursuing actively a Ph.D. thesis on the use of semantic techniques to accelerate and secure Java technologies. He holds an M.Sc. degree from Laval University. In the past years, Hamdi Yahyaoui worked on the formal dynamic semantics of Java and also on the control flow analysis of Java. He participated in the implementation of an LSFM Java optimizing compiler.
e-mail: hamdi@ift.ulaval.ca
LSFM Research Group, Computer Science Department
Laval University
Quebec, G1K 7P4, Canada

INFORMATION FOR AUTHORS

The *Journal of Telecommunications and Information Technology* is published quarterly. It comprises original contributions, both regular papers and letters, dealing with a broad range of topics related to telecommunications and information technology. Items included in the journal report primary and/or experimental research results, which advance the base of scientific and technological knowledge about telecommunications and information technology.

The *Journal is* dedicated to publishing research results which advance the level of current research or add to the understanding of problems related to modulation and signal design, wireless communications, optical communications and photonic systems, speech devices, image and signal processing, transmission systems, network architecture, coding and communication theory, as well as information technology. Suitable research-related manuscripts should hold the potential to advance the technological base of telecommunications and information technology. Tutorial and review papers are published by invitation only.

Papers published by invitation and regular papers should contain up to 15 and 8 printed pages respectively (one printed page corresponds approximately to 3 double-space pages of manuscript, where one page contains approximately 2000 characters).

Manuscript: An original and two copies of the manuscript must be submitted, each completed with all illustrations and tables attached at the end of the papers. Tables and figures have to be numbered consecutively with Arabic numerals. The manuscript must include an abstract limited to approximately 100 words. The abstract should contain four points: statement of the problem, assumptions and methodology, results and conclusion, or discussion, of the importance of the results. The manuscript should be double-spaced on only one side of each A4 sheet (210 × 297 mm). Computer notation such as Fortran, Matlab, Mathematica etc., for formulae, indices, etc., is not acceptable and will result in automatic rejection of the manuscript. The style of references, abbreviations, etc., should follow the standard IEEE format.

References should be marked in the text by Arabic numerals in square brackets and listed at the end of the paper in order of their appearance in the text, including exclusively publications cited inside. The **reference entry** (correctly punctuated according to the following rules and examples) **has to contain**.

From journals and other serial publications: initial(s) and second name(s) of the author(s), full title of publication (transliterated into Latin characters in case it is in Russian, possibly preceded by the title in Russian characters), appropriately abbreviated title of periodical, volume number, first and last page number, year. E.g.:

[1] Y. Namihira, „Relationship between nonlinear effective area and modefield diameter for dispersion shifted fibres", *Electron. Lett.*, vol. 30, no. 3, pp. 262-264, 1994.

From non-periodical, collective publications: as above, but after title – the name(s) of editor(s), title of volume and/or edition number, publisher(s) name(s) and place of edition, inclusive pages of article, year. E.g.:

[2] S. Demri, E. Orłowska, „Informational representability: Abstract models versus concrete models" in *Fuzzy Sets, Logics and Reasoning about Knowledge*, D. Dubois and H. Prade, Eds. Dordrecht: Kluwer, 1999, pp. 301-314.

From books: initial(s) and name(s) of the author(s), place of edition, title, publisher(s), year. E.g.:

[3] C. Kittel, *Introduction to Solid State Physics*. New York: Wiley, 1986.

Figure captions should be started on separate sheet of papers and must be double-spaced.

Illustration: Original illustrations should be submitted. All line drawings should be prepared on white drawing paper in black India ink. Drawings in Corel Draw and Postscript formats are preferred. Colour illustrations are accepted only in exceptional circumstances. Lettering should be large enough to be readily legible when drawing is reduced to two- or one-column width – as much as 4:1 reduction from the original. Photographs should be used sparingly. All photographs must be gloss prints. All materials, including drawings and photographs, should be no larger than 175 × 260 mm.

Page number: Number all pages, including tables and illustrations (which should be grouped at the end), in a single series, with no omitted numbers.

Electronic form: A floppy disk together with the hard copy of the manuscript should be submitted. It is important to ensure that the diskette version and the printed version are identical. The diskette should be labelled with the following information: a) the operating system and word-processing software used, b) in case of UNIX media, the method of extraction (i.e. tar) applied, c) file name(s) related to manuscript. The diskette should be properly packed in order to avoid possible damage during transit.

Among various acceptable word processor formats, $\TeX$ and $\LaTeX$ are preferable. The *Journal's* style file is available to authors.

Galley proofs: Proofs should be returned by authors as soon as possible. In other cases, the article will be proofread against manuscript by the editor and printed without the author's corrections. Remarks to the errata should be provided within two weeks after receiving the offprints.

The *copy of the „Journal"* shall be provided to each author of papers.

Copyright: Manuscript submitted to this journal may not have been published and will not be simultaneously submitted or published elsewhere. Submitting a manuscript, the authors agree to automatically transfer the copyright for their article to the publisher if and when the article is accepted for publication. The copyright comprises the exclusive rights to reproduce and distribute the article, including reprints and also all translation rights. No part of the present journal may be reproduced in any form nor transmitted or translated into a machine language without permission in written form from the publisher.

Biographies and photographs of authors are printed with each paper. Send a brief professional biography not exceeding 100 words and a gloss photo of each author with the manuscript.

National Institute
of Telecommunications
Szachowa st 1
04-894 Warsaw, Poland

Editorial Office

tel. +48(22) 872 43 88
tel./fax:+48(22) 512 84 00
e-mail: redakcja@itl.waw.pl
http://www.itl.waw.pl/jtit