Digital signature in automatic analyses for confidentiality against active adversaries

Ilja Tshahhirov
Tallinn University of Technology

Peeter Laud
Tartu University and Cybernetica AS

Abstract

In this article we enhance the technique of static analysis for confidentiality in cryptographic protocols with support for digital signature operations. The presented technique is an extension of cryptographic protocol analysis presented by Laud, being similar to Abadi and Rogaway — based on replacing the cryptographic operations in the protocols with constructs that are "obviously secure". The replacements are made in such a way that no insecure protocol becomes secure. The transformed protocols are then statically analysed; they should be easier to analyse than the original protocol. Handling the digital signatures is a step in exploring the general approach and making it able to handle realistic scenarios gracefully.

1 Introduction

A cryptographic protocol is expected to satisfy certain security properties. One of them is confidentiality — whether the adversary is able to gain some information about the secret messages transferred during the execution of the protocol.

There are techniques for automatic checking the confidentiality of the protocols. Our work is an extension of analysis presented by Laud [11] to the digital signature primitive. Our main contribution is extending the programming language of protocols analysed in [11] with operations for generating and verifying digital signatures, extending and adjusting the rest of the analysis appropriately. The correctness of this transformation is based on the security definition of signature scheme. This contribution gives a completely automatable method of analysing protocols for the preservation of confidentiality of secret messages, the results of the analysis are correct in the computational model, even when considering *active* adversaries.

This paper has the following structure. After reviewing some related work in Sec.2 we state the security properties of the considered signature scheme in Sec. 3 and extend the protocol language in Sec. 4. We give the security definition of the protocol in Sec. 5. In Sec. 6 we describe our main contribution — a protocol analysis extension to digital signature primitive. Last, we describe a very simple and conservative information flow analysis in Sec. 7 that is nevertheless quite successful in analysing the protocols. We give an example of analysis in Sec. 8.

2 Related Work

The results presented in this paper belong to a body of work attempting to reconcile the two main approaches for modelling and analysing cryptographic protocols — the Dolev-Yao (or "formal") model [8] and the complexity-theory-based (or "computational") approach [15]. Indeed, the semantics of our protocols is defined in terms of the computational model, while the analysis is closer to the formal model. The work in this area has been started by Abadi and Rogaway [1] who considered the relationship of formal and computational symmetric encryption under passive attacks. The same primitive and class of attacks has been further considered in [2, 14, 10, 13, 3], in these papers the language has been expanded (the constraints have been weakened) and the security definitions have been clarified.

If active adversaries are considered as well then the reconciliation approaches have mostly considered asymmetric primitives and/or integrity properties (i.e. properties of execution traces). Hence the digital signatures are one of the most-investigated primitives in this setting. Note that it only makes sense to consider digital signatures in settings where the adversary is active because a passive adversary cannot forge anything. Several implementations of digital signatures and related functionalities (certification, etc.) have been proposed in the reactive simulatability / universal composability framework [4, 5, 6]. Both the reactive and non-reactive frameworks attempt to analyse a protocol in two steps. The first step is to translate the protocol to a form that is more similar to the Dolev-Yao model and hence more easily analysed; and the second step performs the actual analysis. The second step can provide no information to the first. In contrast, the framework that we use [11] performs both steps in parallel, thereby having the potential to achieve better precision.

3 Basic Cryptographic Notions

A function $f : \mathbb{N} \to \mathbb{R}$ is *negligible* if for all positive polynomials p there exists $n_0 \in \mathbb{N}$ such that for all $n \geq n_0$, $|f(n)| < 1/p(n)$. The cryptographic algorithms work with bit-strings, we denote the set of all bitstrings with Σ.

A signature scheme is a triple of polynomial-time algorithms ($\mathcal{G}$, $\mathcal{S}$, $\mathcal{T}$). $\mathcal{G}$ and $\mathcal{S}$ are probabilistic, $\mathcal{T}$ is deterministic. Here $\mathcal{G}$ is the *key pair generation algorithm*, $\mathcal{S}$ is the *signature generation algorithm*, and $\mathcal{T}$ is the *signature verification algorithm.* All algorithms take the *security parameter* n (represented in unary — 1^n) as an argument. Additionally, $\mathcal{S}$ takes two more arguments — secret signing key and message to sign. Also, $\mathcal{T}$ takes three more arguments — public test key, signature, and message to verify this signature with.

Let $msg \in \Sigma$. For all pairs (sk,pk) that may be returned by $\mathcal{G}$ and for all signatures s that may be returned by $\mathcal{S}(1^n, sk, msg)$, the algorithm $\mathcal{T}(1^n, pk, s, msg)$ must return true. This definition only states that $\mathcal{T}$ should recognize signatures, correctly generated with $\mathcal{S}$. We also want the signature system to be secure, i.e. forging the signature without possessing the secret key should be hard.

The signature oracle $\mathcal{O}_{sig}(\cdot)$ used in the security definition below is defined as follows (the security parameter n is assumed to be fixed):

- During the initialization, $\mathcal{O}_{sig}$ generates the key pair (sk, pk) by invoking the $\mathcal{G}(1^n)$, stores it in memory, and outputs the public key pk.
- Upon request to sign the message m, $\mathcal{O}_{sig}(m)$ produces the signature by invoking the $\mathcal{S}(1^n, sk, m)$ (the sk stored during the initialization is used) and returns it.

The signature scheme $(\mathcal{G}, \mathcal{S}, \mathcal{T})$ is called existentially unforgeable under adaptive chosen message attack (ACMA) if for every probabilistic polynomial-time (PPT) machine A_{sig} that interacts with the signature oracle $\mathcal{O}_{sig}(\cdot)$ based on this scheme the following holds: The probability is negligible (in n) that A_{sig} finally outputs two values m and s (meant as a forged signature for the message m) with $\mathcal{T}(1^n, pk, s, m) = \mathsf{true}$ and where m is not among the messages previously signed by $\mathcal{O}_{sig}(\cdot)$.

Both these definitions are similar to the ones used in [5], with the only difference that the maximum number of signatures ns that can be generated with key pair is passed to the signature oracle as an intialization parameter in [5]. As the number of signatures generated during the execution of the protocol subject to the analysis is bounded (one of the key assumptions in the original analysis - [11]), we assume that ns is sufficiently large and omit ns from further consideration.

One more property we require for this analysis is that the signature generation algorithm should return different signatures each time (this property is required in order to analyse all the signatures separately while replacing the signature test operations). It can be formalized as follows. Consider the oracle $\mathcal{S}(1^n, \cdot, \cdot)$. This oracle invokes the signing algorithm taking both the message to be signed and the secret key as arguments. We require that no adversary can cause (with non-negligible probability) the oracle $\mathcal{S}(1^n, \cdot, \cdot)$ to give the same answer to two queries.

A signature scheme satisfying the last requirement can be constructed. Let $(\mathcal{G}', \mathcal{S}', \mathcal{T}')$ be any signature scheme existentially unforgeable against ACMA (such systems exist under standard assumptions, see [5]). We construct the signature scheme $(\mathcal{G}, \mathcal{S}, \mathcal{T})$ as follows:

- $\mathcal{G}$ is equal to $\mathcal{G}'$.
- $\mathcal{S}(1^n, sk, m)$ first generates the signature s by invoking the $\mathcal{S}'(1^n, sk, m)$. Then, it generates a random bit-string r of length n and outputs (s, r).

- $\mathcal{T}(1^n, pk, (s,r), m)$ just returns the result of $\mathcal{T}'(1^n, pk, s, m)$. Note that r is thrown away. As we do not assume that adversary is unable to generate another valid signature for already signed message, the only function of r is to randomize the generated signature.

Obviously the probability that $\mathcal{S}$ returns the same answer twice is bounded from above by $q^2/2^n$, where q is the number of queries that the adversary makes to $\mathcal{S}(1^n, \cdot, \cdot)$. It is negligible, because q is at most polynomial in n. It is also obvious that the resulting signature scheme remains existentially unforgeable against ACMA.

4 Syntax and semantics of a protocol

Protocol, as it is understood here, is a set of programs performing certain computations over the values known to them (computed or received over the network) and exchanging messages. Additionally, there are long term secrets known or shared by participants, initialized prior to the start of the protocol. Let the set **Var** include all the variables used in the protocol.

$$\begin{array}{rlll} P ::= & k := \mathbf{gen\_key} & \mid\ y := (x_1, \ldots, x_m) & \mid\ x := \pi_i^m(y) \\ \mid & x := encr_k(y) & \mid\ y := decr_k(x) & \mid\ x := \mathbf{random} \\ \mid & \mathbf{send}\ x & \mid\ x := \mathbf{receive}^\ell & \mid\ \mathbf{check}(x = y) \\ \mid & x := \mathbf{constant}(b) & \mid\ x := y & \mid\ kp := gen\_key\_pair \\ \mid & pk := public\_key(kp) & \mid\ sm := sign_{kp}(m) & \mid\ test_{pk}(sm) \\ \mid & m := get\_signed\_msg(sm) & & \end{array}$$

$k, x, x_1, \ldots, x_m, y, kp, pk, m, sm$ are *variables* from a set **Var** and b is a function mapping the security parameter to some bit-string.

Figure 1: Statements

A program is a sequence of statements. The statements allowed in the programs are defined in Fig. 1. The semantics of the most of the statements is trivial. $kp := gen\_key\_pair$ generates public-secret key pair, which may be used for signing the messages. $pk := public\_key(kp)$ extracts public key from the key pair. $sm := sign_{kp}(m)$ generates signature for the message and returns the signed message. Both $public\_key(kp)$ and $sign_{kp}(m)$ get stuck (the execution of correspondig participant is stopped) if kp is not valid key pair. $m := get\_signed\_msg(sm)$ extracts the message itself from the signed message; gets stuck if sm is not signed message. $test_{pk}(sm)$ gets stuck if sm was not correctly produced by $sm := sign_{kp}(m)$ (pk is $public\_key(kp)$); it also gets stuck if sm is not signed message or kp is not valid public key.

More detailed information on the operations not related to the digital signature primitive can be found in the [11]. Their *modus operandi* is similar — they get stuck if something is wrong. In particular, the i-th projection π_i^m gets stuck if its argument is not a m-tuple and $\mathbf{check}(x = y)$ gets stuck if the values of x and y differ. The superscript ℓ of a **receive**-statement denotes the expected length (as a function of the security parameter) of the incoming message.

In a protocol, each variable is allowed to occur at most once at the left side of an assignment. Let $\mathbf{Var}_{\mathrm{sent}}$ be the set of all variables x, such that the statement **send** x occurs in the protocol.

The intermediate protocols resulting from the transformation may include some more statements: semaphores, case-statements, and $in()$-statements.

Semaphores (simple synchronization primitives) are implemented using statements $signal(s)$ and $wait(s)$ ($s \in \mathbf{Sem}$ is a semaphore). The statement $signal(s)$ records that the semaphore s has been passed and the statement $wait(s)$ checks whether s has been passed and becomes stuck, if this is not the case. Each semaphore s may occur at most once in the statement $signal(s)$. The format and semantics of *case*-statements are defined in [11]. The $in_z(x_1, \ldots, x_n)$-statement checks whether z is equal to one of the x_i ($i = 1 \ldots n$). If $z = x_i$ ($i = 1 \ldots n$), the execution continues. Otherwise, it gets stuck.

4.1 Protocol execution

We now define what is done during the protocol execution. The emphasis is on the execution of statements related to the digital signature, other statements are described very briefly. [11] contains full details on them.

The following notions are used in the description of the protocol execution: **Parts** — set of (the names of) protocol participants; $\wp(\mathsf{pre})$ — protocol prelude, $\mathbf{Parts}'$ — $\mathbf{Parts}\dot\cup\{\mathsf{pre}\}$, $\wp$ — protocol — function mapping each element of $\mathbf{Parts}\dot\cup\{\mathsf{pre}\}$ to a program.

The protocol runs in parallel to the adversary, which schedules the participants and relays messages between them. Let **ASt** be the set of all possible internal states of the adversary; the set **ASt** is (isomorphic to) some subset of Σ. Adversary $\mathcal{A}(1^n, \ldots)$ implements a probabilistic function $\mathcal{A}_n$ from $\Sigma_\perp \times \mathbf{ASt}$ to $\Sigma \times \mathbf{Parts}' \times \mathbf{ASt}$; the running time of $\mathcal{A}(1^n, \ldots)$ must be polynomial in n. Here n is security parameter, and $X_\perp$ is the set $X \dot{\cup} \{\perp\}$. The $\mathcal{A}_n$ takes the message sent to the network, and returns the message received by a participant executing $x :=$ **receive** at the next moment, as well as the name of the protocol participant that executes next.

A fixed variable $M \in \mathbf{Var}$ contains secret message before the execution of the rest of the protocol. Other variables get their values during the run of the protocol. Let p be a polynomial and let τ be an efficiently computable injective function from $\Sigma \times \Sigma$ to Σ, such that τ^{-1} is also efficiently computable. Let $(\mathcal{G}, \mathcal{S}, \mathcal{T})$ be a fixed secure signature scheme. Let $\mathsf{keypair}$, $\mathsf{publickey}$, and $\mathsf{signedmessage}$ be certain fixed bit-strings. The execution of the protocol proceeds as follows.

Initialisation.

The *execution state* E is the sequence of all the values describing the running state of the protocol. E contains the values of all initialized variables, but also other quantities like the parts of programs that are yet unexecuted, the currently executing participant, the message currently in transit (between a participant and the adversary), and the current state of the adversary. Let **ExcSt** be the set of all execution states.

During the protocol initialization the initial values to the components of E are set. It is done exactly as in the original analysis [11].

Step. During a protocol step, a statement at the head of one of the currently unexecuted parts of protocols gets executed. Let S be this statement. Here we define what happens if S is signature-related. Please refer to [11] for the rest.

- If S is $kp := gen\_key\_pair$ then invoke $(sk, pk) := \mathcal{G}(1^n)$, set $v_{kp} := \tau(\mathsf{keypair}, \tau(sk, pk))$. Also, set i_{kp} to true. Here v_{kp} is the value of the variable kp and i_{kp} is a flag showing whether kp has been initialized.
- If S is $pk := public\_key(kp)$, then parse v_{kp} as $(t, x_1) := \tau^{-1}(kp)$, and $(\_, x_3) := \tau^{-1}(x_1)$. If some invocation of τ^{-1} fails or $t \neq \mathsf{keypair}$, set r_C to false. Otherwise set v_{pk} to $\tau(\mathsf{publickey}, x_3)$ and i_{pk} to true. Here C is the name of the currently running participant and r_C is a flag showing whether C has not become stuck yet.
- If S is $sm := sign_{kp}(msg)$ then parse v_{kp} as $(t, x_1) := \tau^{-1}(kp)$, and $(sk, \_) := \tau^{-1}(x_1)$. If some invocation of τ^{-1} fails or $t \neq \mathsf{keypair}$, set r_C to false. Otherwise, set v_{sm} to $\tau(\mathsf{signedmessage}, \tau(v_{msg}, \mathcal{S}(1^n, sk, v_{msg})))$ and i_{sm} to true.
- If S is $test_{pk}(sm)$ then parse v_{pk} as $(t_1, pk) := \tau^{-1}(v_{pk})$, v_{sm} as $(t_2, x) := \tau^{-1}(v_{sm})$, and $(m, s) := \tau^{-1}(x)$. If any invocation of τ^{-1} fails, or $t_1 \neq \mathsf{publickey}$, or $t_2 \neq \mathsf{signedmessage}$, set r_C to false. Otherwise, set r_C to the value returned by $\mathcal{T}(1^n, pk, s, m)$.
- If S is $msg := get\_signed\_msg(sm)$, then parse v_{sm} as $(t, x) := \tau^{-1}(v_{sm})$ and $(m, \_) := \tau^{-1}(x)$. If any invocation of τ^{-1} fails, or $t \neq \mathsf{signedmessage}$, set r_C to false. Otherwise, set v_{msg} to m and i_{msg} to true.
- If S is $in_z(x_1, \ldots, x_n)$ and i_z is false, set r_C to false. Otherwise, find the first index j, such that i_{x_j} is true and $v_{x_j} = v_z$. If there is no such such j or i_{x_j} is false then set r_C to false. Otherwise, continue execution.

The steps define a probabilistic transition system over the set of execution states. For each $E \in \mathbf{ExcSt}$ there's certain probability that execution of the protocol (together with the adversary) stops at this step. This probability distribution over the **ExcSt** (together with adversary $\mathcal{A}$) is denoted by the $[\![\wp, \mathcal{A}]\!]_n$.

Fixing the random coin tosses made during the execution of each protocol statement and the adversary gives us a single state where the execution stops. If those random coin tosses are defined by $\mathbf{r}$ (as defined in [11]), this final execution state is denoted by $[\![\wp, \mathcal{A}]\!]_n^{\mathbf{r}}$.

4.2 An example

To illustrate our language, consider the simple protocol — transferring the public key and signed message with verification of the signature followed by that. Note that public key is explicitly made known to the adversary. This protocol is shown in Fig. 2.

Protocol prelude	the program for A	the program for B
$K_{AB} := \mathbf{gen\_key}^{(1)}$	$KP_A := gen\_key\_pair^{(2)}$ $PK_A := public\_key(KP_A)$ **send** PK_A $PK_e := encr_{K_{AB}}(PK_A)^{(3)}$ **send** PK_e $SM := sign_{KP_A}(M)^{(4)}$ $SM_A := encr_{K_{AB}}(SM)^{(5)}$ **send** SM_A	$m_1 := \mathbf{receive}^{l_1(6)}$ $PK_B := decr_{K_{AB}}(m_1)$ $m_2 := \mathbf{receive}^{l_2(7)}$ $SM_B := decr_{K_{AB}}(m_2)$ $test_{PK_B}(SM_B)$

Figure 2: Example protocol $\wp$ in our language

5 Security definition

A probability distribution over some finite or countable set X is a function $\mu : X \to [0,1]$, such that $\sum_{x\in X} \mu(x) = 1$. We denote the set of all probability distributions over the set X by $\mathcal{D}(X)$. We denote picking the value of the random variable x according to the distribution $D \in \mathcal{D}(X)$ by $x \leftarrow D$. The notation $\{\!|E : C|\!\}$ denotes the distribution of the random variable or expression E with the probability space defined by C, here C is a list of picking values to random variables and of defining variables.

The definition of the protocol security has not been changed from the original analysis - the protocol is considered secure if there's no non-negligible correlation between the final state of the adversary and the secret message:

$$\{\!|(E[v_M], E[AS]) : E \leftarrow [\![\wp, \mathcal{A}]\!]_n|\!\}_n \approx \{\!|(E'[v_M], E[AS]) : E, E' \leftarrow [\![\wp, \mathcal{A}]\!]_n|\!\}_n \ . \qquad (1)$$

The v_M is the value of secret message and the AS is the final state of the adversary.

The exact meaning of indistinguishability between two *families* of probability distributions $D = \{D_n\}_{n\in\mathbb{N}}$ and $D' = \{D'_n\}_{n\in\mathbb{N}}$, denoted $D \approx D'$, is following: for all PPT algorithms $\mathcal{A}$, the difference of probabilities

$$\mathbf{P}[b = 1 \,|\, x \leftarrow D_n, b \leftarrow \mathcal{A}(1^n, x)] - \mathbf{P}[b = 1 \,|\, x \leftarrow D'_n, b \leftarrow \mathcal{A}(1^n, x)]$$

is a negligible function of n.

6 Analysing the protocols

The analysis of signature operations is done in several steps. First, the protocol is abstractly interpreted - each variable is assigned a term describing the construction of its value from the atomic values (keys, key pairs, public keys, signed messages, random numbers, constants, secret message, and advisory's input) - this is necessary for tracking the usage of key pairs. This part is described in the Sec. 6.1. Given the abstract interpretation, the protocol is analysed for the dead code, as described in Sec. 6.2. Then we attempt to find the key pairs which are not made available to the adversary (this check is necessary, because exposing the secret key ruins the definition of the signature scheme security). After the candidate key pair is found, we attempt to do a modification to the protocol, directly following from the security properties of the signature scheme. Given that without possessing the secret key it is computationally infeasible to generate valid signature for the message that has not been previously signed, we can make certain assumptions on the value of the message being tested. In order for $test_{pk}(sm)$ operation to succeed, sm has to be equal to one of the messages signed with the corresponding key pair, so we can supply the additional axioms. These additional axioms serve two purposes: first help us in statically analysing the outcome of $test_{pk}(sm)$ operation, and then they allow taking into the account the authenticity of the signed data (which may be useful in further extension of the analysis, for instance, to the public key cryptography, where the public key may be sent unencrypted, but signed). The axioms are introduced in the form of $in_{sm}(sm_1, \ldots, sm_n)$, where $sm_1, \ldots, sm_n$ are all the messages which have been signed with the corresponding key pair. Then, the $in()$-statement is analysed in the same way as *case*-statement - by splitting the original protocol $\wp$ into branches $\wp_1, \ldots, \wp_n$, each corresponding to one of the possible values checked against in the $in()$-statement. Finding the key pairs, as well as transforming the protocol is described in Sec. 6.3.

After the modifications are done, the information flow analysis, described in Sec. 7, is applied to the protocol(s). The process of analysis is iterative — modifications are done until either protocol can be considered secure or no more transformations are possible — in this case the protocol is considered potentially insecure.

6.1 Interpreting the protocol

Let all the statements in the protocol be labelled by elements of set **Lab**. Different statements must have different labels. For all $A \in \mathbf{Parts}'$, let l_A be the length of the program $\wp(A)$. Let $L_\wp$ (used to enumerate all the statements in the protocol) be the following set:

$$L_\wp = \{(A, i) \mid A \in \mathbf{Parts}', 1 \leq i \leq l_A\} \ . \tag{2}$$

Terms are used to track how the values of the variables are constructed from the "atomic" values. The definition of terms $T \in \mathbf{Trm}$ is defined in Fig. 3.

$$\begin{array}{rlllll} T & ::= & \underline{key}(l) & \mid \underline{random}(l) & \mid \underline{const}(b) & \mid \underline{secret} \\ & \mid & \underline{tuple}^m(T_1, \ldots, T_m) & \mid \underline{\pi}_i^m(T_1) & \mid \underline{input}(l) & \mid \underline{decr}(T_1, T_2) \\ & \mid & \underline{encr}(l, T_1, T_2) & \mid \underline{keypair}(l) & \mid \underline{publickey}(T_1) & \mid \underline{signedmsg}(l, T_1, T_2) \\ & \mid & \underline{msg}(T_1) & & & \end{array}$$

Figure 3: The set of abstract values

Let $\mathsf{I}(x) \in \mathbf{Trm}$ denote the term assigned to the variable x. We let $\mathsf{I}(M) = \underline{secret}$, where $M \in \mathbf{Var}$ is the variable containing the secret value. The rest of the terms $\mathsf{I}(x)$ are defined by examining the statement defining x. The definition of $\mathsf{I}(x)$ for signature-related terms is given in Fig. 4 (for other terms, refer to [11]). Note that there is no term for *test*(), as this statement does not affect the values of

$x := \ldots$	$\mathsf{I}(x)$ is
$gen\_key\_pair^l$	$\underline{keypair}(l)$
$public\_key(kp)$	$\underline{publickey}(\mathsf{I}(kp))$
$sign_{kp}(m)^l$	$\underline{signedmsg}(l, \mathsf{I}(kp), \mathsf{I}(m))$
$get\_signed\_msg(sm)$	$\underline{msg}(\mathsf{I}(sm))$

Figure 4: Definition of the terms $\mathsf{I}(x)$

the variables. Also note that the label of the statement is used as the term argument for those operations which depend on either random coin tosses or adversary actions. That is necessary in order to be able to analyse two instances of the protocol executing in lock-step (with fixed coin-tosses).

The semantics of the terms $T \in \mathbf{Trm}$ are defined by $\iota_n^{\mathbf{r}} : \mathbf{Trm} \to \Sigma_\perp$ (the random choices $\mathbf{r} \in \mathbf{R}$ are fixed):

- $\iota_n^{\mathbf{r}}(\underline{keypair}(l)) = \tau(\mathsf{keypair}, \tau(sk, pk))$, where (sk, pk) is generated by invocation of $\mathcal{G}^{\mathbf{r}(l,\mathsf{p})}(1^n)$. Here $\mathbf{r}(l, \mathsf{p})$ denotes the random coins in $\mathbf{r}$ intended for the legitimate participant at the statement labelled by l. These coins are used as the random coins for the key generation algorithm. Also, the generated key pair is *tagged with* keypair.
- $\iota_n^{\mathbf{r}}(\underline{publickey}(T))$ is $\perp$ (meaning that the statement defining the variable will get stuck) if $\iota_n^{\mathbf{r}}(T)$ is $\perp$. Otherwise, T is parsed as follows: $(t_1, kp) := \tau^{-1}(\iota_n^{\mathbf{r}}(T))$ and $(\_, pk) := \tau^{-1}(kp)$. If t_1 is not keypair, or any invocation of τ^{-1} fails, $\iota_n^{\mathbf{r}}(\underline{publickey}(T))$ is $\perp$. Otherwise, it is equal to $\tau(\mathsf{publickey}, pk)$.
- $\iota_n^{\mathbf{r}}(\underline{signedmsg}(l, T_1, T_2))$ is $\perp$ if $\iota_n^{\mathbf{r}}(T_1)$ is $\perp$ or is not tagged with keypair or $\iota_n^{\mathbf{r}}(T_2)$ is $\perp$. Otherwise it is equal to $\tau(\mathsf{signedmessage}, \tau(\iota_n^{\mathbf{r}}(T_2), \mathcal{S}^{\mathbf{r}(l,\mathsf{p})}(1^n, t_1, \iota_n^{\mathbf{r}}(T_2))))$ where $(\_, kp) = \tau^{-1}(\iota_n^{\mathbf{r}}(T_1))$ and $(t1, \_) = \tau^{-1}(kp)$. If any of the invocations of τ^{-1} fails, the semantics of $\iota_n^{\mathbf{r}}(\underline{signedmsg}(l, T_1, T_2))$ is $\perp$.
- $\iota_n^{\mathbf{r}}(\underline{msg}(T))$ is computed as follows: $(t_1, x) = \tau^{-1}(\iota_n^{\mathbf{r}}(T))$ and $(t_2, \_) = \tau^{-1}(x)$. If t_1 is not signedmessage, or any invocation of τ^{-1} fails, $\iota_n^{\mathbf{r}}(\underline{msg}(T))$ is $\perp$. Otherwise, it is equal to t_2.

Let $X_n^{\mathbf{r}} \subseteq L_\wp$ denote the set of statements that are actually executed (for the random choices $\mathbf{r}$), i.e. the execution of the protocol is not stopped before reaching that statement and the participant containing that statement does not get stuck at or before that statement.

Theorem 1. *Let $x \in \mathbf{Var}$ and let $l \in L_\wp$ be (the label of) the statement defining x. If $l \in X_n^{\mathbf{r}}$ or $x = M$ then*

$$[\![\wp, \mathcal{A}]\!]_n^{\mathbf{r}}[v_x] = \iota_n^{\mathbf{r}}(\mathsf{I}(x)) \ . \tag{3}$$

Theorem is proven by induction over the order of executing the statements (determined by $\mathbf{r}$). It mainly consists of comparing the definitions of the execution step and $\iota_n^{\mathbf{r}}$. Please refer to [11] for it.

There are rules for deriving the equality and inequality of two terms, denoted by $T_1 \simeq T_2$ and $T_1 \not\simeq T_2$, correspondingly. For example, getting the message body and signing the message cancel out (under certain conditions). An example of inequality is that different key pairs are unequal. Also, if the semantics of a certain term is almost always $\perp$, we denote it by $\mathsf{stuck}\, T$.

Fig. 5 presents the axioms and inference rules defining $\simeq$, $\not\simeq$ and stuck for terms introduced due to or affected by the inclusion of digital signature primitive (complete set of rules can be found in [11]). The syntax of the rules is quite simple - if the premises (above the horizontal line) are satisfied, then the rule consequent (below the horizontal line) also holds. For instance, the rule (nec) says that the type of the value that a bit-string represents can be determined from that bit-string. The rules (dif$_*$) say that different random bit-strings are different. The rules (inc$_*$) state that the corresponding operations check the types of their inputs.

$$\frac{\underline{C}, \underline{C'} \in \{\underline{key}, \underline{random}, \underline{const}, \underline{secret}, \underline{tuple}^n, \underline{encr}, \underline{publickey}, \underline{signedmsg}, \underline{keypair}\} \quad \underline{C} \neq \underline{C'}}{\underline{C}(\ldots) \not\simeq \underline{C'}(\ldots)}\,(\text{nec})$$

$$\frac{l \neq l'}{\underline{keypair}(l) \not\simeq \underline{keypair}(l')}\,(\text{dif}_{\text{kp}}) \qquad \frac{\underline{C} \in \{\underline{key}, \underline{random}, \underline{const}, \underline{tuple}^n, \underline{encr}, \underline{publickey}, \underline{signedmsg}\}}{\mathsf{stuck}\ \underline{publickey}(\underline{C}(\ldots))}\,(\text{inc}_{\text{pk}})$$

$$\frac{l \neq l'}{\underline{signedmsg}(l, T_1, T_2) \not\simeq \underline{signedmsg}(l', T_3, T_4)}\,(\text{dif}_{\text{s}}) \qquad \frac{\underline{C} \in \{\underline{keypair}, \underline{publickey}, \underline{signedmsg}\}}{\mathsf{stuck}\ \underline{\pi_i^m}(\underline{C}(\ldots))}\,(\text{inc}_{\text{p2}})$$

$$\frac{\underline{C} \in \{\underline{key}, \underline{random}, \underline{const}, \underline{tuple}^n, \underline{encr}, \underline{publickey}, \underline{signedmsg}\}}{\mathsf{stuck}\ \underline{signedmsg}(\underline{C}(\ldots), T_1)}\,(\text{inc}_{\text{s}})$$

$$\frac{\underline{C} \in \{\underline{keypair}, \underline{publickey}, \underline{signedmsg}\}}{\mathsf{stuck}\ \underline{decr}(T, \underline{C}(\ldots))}\,(\text{inc}_{\text{e3}}) \qquad \frac{\underline{C} \in \{\underline{keypair}, \underline{publickey}, \underline{signedmsg}\}}{\mathsf{stuck}\ \underline{decr}(\underline{C}(\ldots), T)}\,(\text{inc}_{\text{e4}})$$

$$\frac{\underline{C} \in \{\underline{key}, \underline{random}, \underline{const}, \underline{tuple}^n, \underline{encr}, \underline{publickey}, \underline{keypair}\}}{\mathsf{stuck}\ \underline{msg}(\underline{C}(\ldots))}\,(\text{inc}_{\text{m}}) \qquad \frac{}{\underline{msg}(\underline{signedmsg}(l, T_{kp}, T_M)) \simeq T_M}\,(\text{can}_{\text{m}})$$

Figure 5: Equality and inequality of terms

We say that $T_1 \simeq T_2$ is *sound at the statement* S^l if the probability

$$\mathbf{P}[l \in X_n^{\mathbf{r}} \wedge \iota_n^{\mathbf{r}}(T_1) \neq \perp \neq \iota_n^{\mathbf{r}}(T_2) \wedge \iota_n^{\mathbf{r}}(T_1) \neq \iota_n^{\mathbf{r}}(T_2) \ : \ \mathbf{r} \in_R \mathbf{R}] \tag{4}$$

is negligible in n. Similarly, $T_1 \not\simeq T_2$ is *sound at the statement* S^l if the probability

$$\mathbf{P}[l \in X_n^{\mathbf{r}} \wedge \iota_n^{\mathbf{r}}(T_1) \neq \perp \neq \iota_n^{\mathbf{r}}(T_2) \wedge \iota_n^{\mathbf{r}}(T_1) = \iota_n^{\mathbf{r}}(T_2) \ : \ \mathbf{r} \in_R \mathbf{R}] \tag{5}$$

is negligible in n. We say that $\mathsf{stuck}\, T$ is *sound at the statement* S^l if the probability

$$\mathbf{P}[l \in X_n^{\mathbf{r}} \wedge \iota_n^{\mathbf{r}}(T) \neq \perp \,|\, \mathbf{r} \in_R \mathbf{R}] \tag{6}$$

is negligible in n. If the statement defining the variable whose abstract value is T is above S then S is dead.

Theorem 2. *All rules in Fig. 5 are sound. I.e. if S^l is a statement of the protocol and all the premises of some rule are sound at S^l then the consequent of that rule is also sound at S^l.*

Proof.

- Rule (nec): the constructors C and C' tag the values that they construct with different tags. The probability of these values being equal is zero.

- Rule (dif_{kp}): The semantics of two terms in the consequent are two different key pairs. There is only negligible chance that they are equal (if the probability of two key pairs being equal were non-negligible then there would be non-negligible chance of guessing key pairs).
- Rule (dif_{s}): The semantics of two terms in the consequent are two different signatures. There is only negligible chance that they are equal (directly follows from the required properties of the encryption scheme).
- Rules (inc_*): clearly, the semantics of the term in the consequent of the rule is always $\perp$.
- Rule (can_{m}): pick $\mathbf{r} \in \mathbf{R}$ in such way that the event described by (4) happens for the premise (if there are any premises). Such picking disallows only a negligible fraction of possible random choices. By definition of $\iota^{\mathbf{r}}_n$, the semantics of a term in the consequence of the rule is either $\perp$ or equal to the semantics of the simpler term. $\square$

6.2 Removing dead code

In addition to removing the cycles in the execution graph (i.e. cases where $\mathit{wait}(s)$ occurs before $\mathit{signal}(s)$) and looking for contradictions from $\simeq$ and $\not\simeq$, as described in [11], one more technique is introduced - checking the types of the arguments to $\mathit{test}()$-statements. The approach is the same as defined for the terms in inc_* rules, however, $\mathit{test}()$-statements do not produce terms, so it needs to be specified separately.

Consider the abstract interpretation of the arguments of statement $\mathit{test}_{pk}(sm)$. If one of the following is true, then the statement and the code following it is dead and can be excluded from further analysis.

- $\mathsf{I}(pk)$ is $C(\ldots)$ where C is one of $\underline{\mathit{key}}$, $\underline{\mathit{random}}$, $\underline{\mathit{const}}$, $\underline{\mathit{tuple}}^n$, $\underline{\mathit{encr}}$, $\underline{\mathit{signedmsg}}$, $\underline{\mathit{keypair}}$.
- $\mathsf{I}(sm)$ is $C(\ldots)$ where C is one of $\underline{\mathit{key}}$, $\underline{\mathit{random}}$, $\underline{\mathit{const}}$, $\underline{\mathit{tuple}}^n$, $\underline{\mathit{encr}}$, $\underline{\mathit{publickey}}$, $\underline{\mathit{keypair}}$.

6.3 Transforming the test-statements

The first task is to choose the key pair which can be replaced by black boxes creating signature with it and getting public key from it - i.e. this key pair may be used only for signing and for public key extraction. Choosing the key pair means selecting $l \in \mathbf{Lab}$, such that $\underline{\mathit{keypair}}(l)$ occurs in the values of variables.

In order to introduce the additional axioms prior the test statements, we have to make sure that adversary is not able to forge signatures (i.e. generate valid signatures that pass test). One conservative way to do that is to make sure that the key pair does not affect the adversary view. Therefore, the rule for $\underline{\mathit{keypair}}(l)$ to be eligible to be chosen is: if the value of some variable x affects the adversary's view, then the $\underline{\mathit{keypair}}(l)$ may occur in $\mathsf{I}(x)$ only in context $\underline{\mathit{signedmsg}}(l, \underline{\mathit{keypair}}(l), T_1)$ or $\underline{\mathit{publickey}}(\underline{\mathit{keypair}}(l))$. Such variables x are:

- the elements of $\mathbf{Var}_{\text{sent}}$;
- the variables occurring in **check**()-statements for which we do not statically know whether the comparison succeeds;
- the variables occurring as ciphertexts in decryption statements;
- the variables occurring as keys in encryption or decryption statements
- the variables occurring as key pairs in signing statements (exception: $\underline{\mathit{keypair}}(l)$ itself may be the abstract value of $\mathsf{I}(x)$);
- the variables occurring as public keys or signed messages in $\mathit{test}()$-statements for which we do not statically know whether the test succeeds or not;
- the variables occurring in the right-hand side of the projection statements for which we do not statically know whether the projection succeeds or not.

While introducing the additional axioms prior to signature tests with public key corresponding to key pair $\underline{\mathit{keypair}}(l)$, we have to make sure that we do not test any signatures that were created with some other key pair. Since the key pair is never sent out (exception - public key component may be sent out - in this case part of the signature test statements — the ones which use public key received from the network —

will not be removed), we know exactly where it is used. Let $sm_1 := sign_{kp_1}(m_1), \ldots, sm_n := sign_{kp_n}(m_n)$ be all statements that $\mathsf{I}(kp_1) \simeq \ldots \simeq \mathsf{I}(kp_n) \simeq \underline{keypair}(l)$. If $test_{pk}(sm)$ is a signature testing statement in the protocol such that $\mathsf{I}(pk) \simeq \underline{publickey}(\underline{keypair}(l))$, then add the following code before it (assuming that tmp is a variable not occurring in the protocol):

$$\begin{array}{l} tmp := get\_signed\_msg(sm) \\ in_{tmp}(m_1, \ldots, m_n) \end{array}$$

By the security property of the signing oracle (existential unforgeable under ACMA), the chances that adversary is able to generate signature that will pass is negligible, so either the $test_{pk}(sm)$ will fail or value of the m will belong to the set of messages signed with the appropriate key pair.

The final step is removing the $in()$. This removal results in one or more protocols, such that the security of all of them implies the security of the original protocol. If the second argument of the $in()$ is an empty set, this statement is a dead code, which (along with the code following it) should be excluded from further analysis.

Let $\mathsf{IN}_1, \ldots, \mathsf{IN}_k$ be the $in()$-statements in the protocol $\wp'$. Let the second argument of the statement IN_i (set) consist of n_i items. The number of protocols $\wp'_1, \wp'_2, \ldots$ will be $n_1 \cdot n_2 \cdots n_k$; they are defined in the following way: for all $i \in \{1, \ldots, k\}$ choose $c_i \in \{1 \ldots, n_i\}$; each of the protocols $\wp'_1, \wp'_2, \ldots$ will correspond to one of such possible choices. Let $s_1, \ldots, s_n \in \mathbf{Sem}$ be semaphores not occurring in the protocol $\wp'$. To obtain a new protocol, we change the $in()$-statement IN_i in the following way:

- Add $signal(s_i)$ immediately after the statement defining x_{c_i}.
- Replace IN_i with the following fragment:

$$\begin{array}{l} wait(s_i) \\ \mathbf{check}(z = x_{c_i}) \end{array}$$

Such change is applied to all $in()$-statements. The resulting protocol has no $in()$-statements. Basically, going from $\wp'$ to $\wp'_1, \wp'_2, \ldots$ just means analysing all choices of the $in()$-statements separately.

We are analyzing the branches separately, therefore we may wonder whether we are missing some implicit flows of information here. In other words, is it possible that all n programs that we get after transforming an $in()$-statement are secure, but the original program is not? Fortunately, this is not possible. In our programming language we have no means to let the value of the secret M influence, which of the branches will be taken. Therefore the choice of a branch only depends on the adversary's actions. If the adversary does not know M before one of the branches is chosen then the chosen branch does not depend on M and hence gives no information about M to the adversary.

When the described transformations are done, the resulting protocols are either submitted to the same analysis (if there still are unprocessed key pairs left) or to the information flow analysis in the next section.

7 A simple information flow analysis

We will now extend a information flow analysis from [11] to determine whether the secret variables (in our case this set is $\{M\}$) affect the view of the adversary.

First, we define the information flow. If some participant of the protocol contains a statement of the form $x{:=}E(x_1, \ldots, x_m)$, where E is any expression and $x_1, \ldots, x_m$ are all variables occurring in it, then we say that there is an *information flow* from the variable x_i to the variable x and write $x_i \Rightarrow x$.

The protocol P is deemed secure if $M \stackrel{*}{\Rightarrow} y$ holds for no y satisfying some of the following conditions:

- $y \in \mathbf{Var}_{\text{sent}}$;
- y occurs in a **check**-statement (unless we are able to determine its result from the inference rules in Fig. 5);
- y occurs as an argument to a projection (π) statement (unless we are able to statically predict whether it succeeds or fails);
- y occurs as a ciphertext in a decryption statement;

- y occurs as a key in an encryption or decryption statement;
- y occurs as an argument to the **test** statement (unless we are able to determine its result from the inference rules);
- y occurs as a key in the **sign** statement.

(here $\overset{*}{\Rightarrow}$ denotes the reflexive transitive closure of $\Rightarrow$). Otherwise the protocol is deemed potentially insecure.

If a protocol is secure then two instances of that protocol which differ only in the value of the secret variable M execute in lock-step, giving the same values to all public parts of the execution context. This result is proven in [11]. Extending it to digital signature operation is also trivial.

8 Analysis example

We demonstrate the developed technique by by applying it to the analysis of the security of the sample protocol from Sec. 4.2 (initial protocol is shown on Fig. 2). The abstract interpretation of that protocol is given in Fig. 6.

$x \in \mathbf{Var}$	$\mathsf{I}(x)$
M	$\underline{secret}$
K_{AB}	$\underline{key}(1)$
KP_A	$\underline{keypair}(2)$
PK_A	$\underline{publickey}(\underline{keypair}(2))$
PK_e	$\underline{encr}(3, \underline{key}(1), \underline{publickey}(\underline{keypair}(2)))$
SM	$\underline{signedmsg}(4, \underline{keypair}(2), \underline{secret})$
SM_A	$\underline{encr}(5, \underline{key}(1), \underline{signedmsg}(4, \underline{keypair}(2), \underline{secret}))$
m_1	$\underline{input}(6)$
PK_B	$\underline{decr}(\underline{key}(1), \underline{input}(6))$
m_2	$\underline{input}(7)$
SM_B	$\underline{decr}(\underline{key}(1), \underline{input}(7))$

Figure 6: Interpretation of protocol $\wp$

Running the information flow analysis, we consider the protocol to be potentially insecure, as $SM_A \in \mathbf{Var}_{\text{sent}}$ and $M \overset{*}{\Rightarrow} SM_A$). Therefore we run protocol transformations in order to simplify the protocol. Since it is a "real" protocol, no dead code is found in it. The first round of transformations is replacing the encryption and decryption statements, as described in [11]. After running it, the protocol looks like shown on Fig.7.

Protocol prelude	the program for A	the program for B
$K_{AB} := \mathbf{gen\_key}^{(1)}$ $Z_1 := \mathbf{constant}(\mathbf{0}^{l_1(n)})$ $Z_2 := \mathbf{constant}(\mathbf{0}^{l_2(n)})$	$KP_A := gen\_key\_pair^{(2)}$ $PK_A := public\_key(KP_A)$ **send** PK_A $PK_e := encr_{K_{AB}}(Z_1)^{(3)}$ **send** PK_e $SM := sign_{KP_A}(M)^{(4)}$ $SM_A := encr_{K_{AB}}(Z_2)^{(5)}$ **send** SM_A	$m_1 := \mathbf{receive}^{l_1(6)}$ $PK_B := case\ m_1\ of$ $PK_e \rightarrow PK_A$ $SM_A \rightarrow SM$ $m_2 := \mathbf{receive}^{l_2(7)}$ $SM_B := case\ m_2\ of$ $PK_e \rightarrow PK_A$ $SM_A \rightarrow SM$ $test_{PK_B}(SM_B)$

Figure 7: Protocol $\wp$ after replacing encryption/decryption

Removing case-statements results in 4 sub-protocols, corresponding to all possible sub-cases (variable assignments). One of the sub-protocols (let's call it $\wp_1$) corresponds to the intended usage of the variables ($PK_B := PK_A$ and $SM_B := SM$). Other cases ($\wp_2$:$PK_B := PK_A$ and $SM_B := PK_A$; $\wp_3$:$PK_B := SM$ and $SM_B := SM$; $\wp_4$:$PK_B := SM$ and $SM_B := PK_A$) will lead to some malfunctioning of the protocol (as values are used in not-intended way).

For illustration purposes, we demonstrate the analysis of $\wp_1$ and $\wp_2$, since the analysis of $\wp_3$ and $\wp_4$ is similar to $\wp_2$.

Protocol prelude	the program for A	the program for B
$K_{AB} :=$ **gen_key**$^{(1)}$ $Z_1 :=$ **constant**$(\mathbf{0}^{l_1(n)})$ $Z_2 :=$ **constant**$(\mathbf{0}^{l_2(n)})$	$KP_A := \mathit{gen\_key\_pair}^{(2)}$ $PK_A := \mathit{public\_key}(KP_A)$ **send** PK_A $PK_e := \mathit{encr}_{K_{AB}}(Z_1)^{(3)}$ $\mathit{signal}(s_1)$ **send** PK_e $SM := \mathit{sign}_{KP_A}(M)^{(4)}$ $SM_A := \mathit{encr}_{K_{AB}}(Z_2)^{(5)}$ $\mathit{signal}(s_2)$ **send** SM_A	$m_1 :=$ **receive**$^{l_1(6)}$ $\mathit{wait}(s_1)$ **check**$(m_1 = PK_e)$ $PK_B := PK_A$ $m_2 :=$ **receive**$^{l_2(7)}$ $\mathit{wait}(s_2)$ **check**$(m_2 = SM_A)$ $SM_B := SM$ $\mathit{test}_{PK_B}(SM_B)$

Figure 8: Protocol $\wp_1$

$x \in$ **Var**	$\mathsf{I}(x)$
Z_1	$\underline{const}(0^{l_1(n)})$
Z_2	$\underline{const}(0^{l_2(n)})$
PK_e	$\underline{encr}(3, \underline{key}(1), \underline{const}(0^{l_1(n)}))$
SM_A	$\underline{encr}(5, \underline{key}(1), \underline{const}(0^{l_2(n)}))$
PK_B	$\underline{publickey}(\underline{keypair}(2))$
SM_B	$\underline{signedmsg}(4, \underline{keypair}(2), \underline{secret})$

Figure 9: Interpretation of protocol $\wp_1$ (changes only)

The abstract interpretation of the protocol $\wp_1$ is given on Fig. 9. The protocol $\wp_1$ is checked for dead code - no dead is code found. At this point, the candidate key pair is found. The only key pair (KP_A) present in the protocol satisfies the criteria. Introducing the additional axioms prior to test statement corresponding to the protocol results in the protocol $\wp_1'$.

Protocol prelude	the program for A	the program for B
$K_{AB} :=$ **gen_key**$^{(1)}$ $Z_1 :=$ **constant**$(\mathbf{0}^{l_1(n)})$ $Z_2 :=$ **constant**$(\mathbf{0}^{l_2(n)})$	$KP_A := \mathit{gen\_key\_pair}^{(2)}$ $PK_A := \mathit{public\_key}(KP_A)$ **send** PK_A $PK_e := \mathit{encr}_{K_{AB}}(Z_1)^{(3)}$ $\mathit{signal}(s_1)$ **send** PK_e $SM := \mathit{sign}_{KP_A}(M)^{(4)}$ $SM_A := \mathit{encr}_{K_{AB}}(Z_2)^{(5)}$ $\mathit{signal}(s_2)$ **send** SM_A	$m_1 :=$ **receive**$^{l_1(6)}$ $\mathit{wait}(s_1)$ **check**$(m_1 = PK_e)$ $PK_B := PK_A$ $m_2 :=$ **receive**$^{l_2(7)}$ $\mathit{wait}(s_2)$ **check**$(m_2 = SM_A)$ $SM_B := SM$ $\mathit{tmp} := \mathit{get\_signed\_msg}(SM_B)$ $\mathit{in}_{\mathit{tmp}}(M)$ $\mathit{test}_{PK_B}(SM_B)$

Figure 10: Protocol $\wp_1'$

Removing the $\mathit{in}()$-statement leads us to the only possible branch, as indicated on Fig. 11.

The abstract interpretation of the protocol $\wp_1'$ with $\mathit{in}()$ removed is given in Fig. 12.

At this point we can conclude that the secret message does not affect adversary view in $\wp_1'$, since, given the interpretations of the SM_B and PK_B, the $\mathit{test}_{PK_B}(SM_B)$ always succeeds. The statement **check**$(\mathit{tmp} = M)$, given the abstract interpretation of tmp, always succeeds either.

Now let us consider the $\wp_2$ - the sub-protocol resulting in "not-intended" information flow. It is shown on Fig.13.

The abstract interpretation of the protocol shown in Fig.13, it is given in Fig. 14.

During the search for dead code we conclude that, given the interpretation of SM_B (which is $\underline{publickey}(\underline{keypair}(2))$), the $\mathit{test}_{PK_B}(SM_B)$ will always get stuck (the type check on SM_B will always fail). Considering the statements preceding it, we conclude that the $\wp_2$ is secure, since according to the information flows analysis M does not affect adversary view.

Sub-protocols $\wp_3$ and $\wp_4$ are analysed similarly to $\wp_2$, and give same result - protocol always gets stuck irrespective of the value of M. So, all sub-protocols are secure. Therefore, the original protocol $\wp$ is secure too.

Protocol prelude	the program for A	the program for B
$K_{AB} := \mathbf{gen\_key}^{(1)}$ $Z_1 := \mathbf{constant}(\mathbf{0}^{l_1(n)})$ $Z_2 := \mathbf{constant}(\mathbf{0}^{l_2(n)})$	$KP_A := \mathit{gen\_key\_pair}^{(2)}$ $PK_A := \mathit{public\_key}(KP_A)$ $\mathbf{send}\ PK_A$ $PK_e := \mathit{encr}_{K_{AB}}(Z_1)^{(3)}$ $\mathit{signal}(s_1)$ $\mathbf{send}\ PK_e$ $SM := \mathit{sign}_{KP_A}(M)^{(4)}$ $SM_A := \mathit{encr}_{K_{AB}}(Z_2)^{(5)}$ $\mathit{signal}(s_2)$ $\mathbf{send}\ SM_A$	$m_1 := \mathbf{receive}^{l_1(6)}$ $\mathit{wait}(s_1)$ $\mathbf{check}(m_1 = PK_e)$ $PK_B := PK_A$ $m_2 := \mathbf{receive}^{l_2(7)}$ $\mathit{wait}(s_2)$ $\mathbf{check}(m_2 = SM_A)$ $SM_B := SM$ $\mathit{tmp} := \mathit{get\_signed\_msg}(SM_B)$ $\mathbf{check}(\mathit{tmp} = M)$ $\mathit{test}_{PK_B}(SM_B)$

Figure 11: Protocol $\wp_1'$ with $\mathit{in}()$ removed

$x \in \mathbf{Var}$	$\mathsf{I}(x)$
tmp	$\underline{\mathit{secret}}$

Figure 12: Interpretation of protocol $\wp_1'$ with $\mathit{in}()$ removed (changes only)

Protocol prelude	the program for A	the program for B
$K_{AB} := \mathbf{gen\_key}^{(1)}$ $Z_1 := \mathbf{constant}(\mathbf{0}^{l_1(n)})$ $Z_2 := \mathbf{constant}(\mathbf{0}^{l_2(n)})$	$KP_A := \mathit{gen\_key\_pair}^{(2)}$ $PK_A := \mathit{public\_key}(KP_A)$ $\mathbf{send}\ PK_A$ $PK_e := \mathit{encr}_{K_{AB}}(Z_1)^{(3)}$ $\mathit{signal}(s_1)$ $\mathbf{send}\ PK_e$ $SM := \mathit{sign}_{KP_A}(M)^{(4)}$ $SM_A := \mathit{encr}_{K_{AB}}(Z_2)^{(5)}$ $\mathit{signal}(s_2)$ $\mathbf{send}\ SM_A$	$m_1 := \mathbf{receive}^{l_1(6)}$ $\mathit{wait}(s_1)$ $\mathbf{check}(m_1 = PK_e)$ $PK_B := PK_A$ $m_2 := \mathbf{receive}^{l_2(7)}$ $\mathit{wait}(s_1)$ $\mathbf{check}(m_2 = SM_A)$ $SM_B := PK_A$ $\mathit{test}_{PK_B}(SM_B)$

Figure 13: Protocol $\wp_2$

9 Conclusion

We have presented the extension of analysis given in [11] to digital signature primitive. The analysis works by first applying transformation described in Sec. 6 to the protocol, resulting in one or more subprotocols. The security of the transformed protocols can be analysed using the information flow analysis given in Sec.7. Our analysis is correct with respect to the computational semantics and (complexity-theoretical) definition of confidentiality.

There are two directions in which this work is planned to be extended:

- Adding the support for another cryptographic primitive — asymmetric encryption.
- Composing an automated prover.

The authors are in progress of doing the work in both of the mentioned directions.

10 Acknowledgement

We are thankful to the anonymous referees of NordSec 2005 conference for their comments. They certainly have improved the readability of this paper.

This research has been supported by the Estonian Information Technology Foundation and the ESF Grants No. 5645 and 6095.

References

[1] Martín Abadi and Phillip Rogaway. Reconciling Two Views of Cryptography (The Computational Soundness of Formal Encryption). International Conference IFIP TCS 2000, LNCS 1872, pages 3–22, August 2000.

$x \in \mathbf{Var}$	$\mathsf{I}(x)$
PK_B	$\underline{publickey}(\underline{keypair}(2))$
SM_B	$\underline{publickey}(\underline{keypair}(2))$

Figure 14: Interpretation of protocol $\wp_2$ (changes relative to $\wp_1$ only)

[2] Martín Abadi and Jan Jürjens. Formal Eavesdropping and its Computational Interpretation. Theoretical Aspects of Computer Software, 4th International Symposium (TACS 2001), LNCS 2215, pages 82–94, September 2001.

[3] Pedro Adão, Gergei Bana, Jonathan Herzog, and Andre Scedrov. Soundness of formal encryption in the presence of key-cycles. 10-th European Symposium on Research in Computer Security (ESORICS 2005), LNCS 3679, pages 374-396, September 2005.

[4] Michael Backes, Birgit Pfitzmann, and Michael Waidner. A Universally Composable Cryptographic Library. 10th ACM Conference on Computer and Communications Security (CCS 2003), October 2003.

[5] Michael Backes, Birgit Pfitzmann, and Michael Waidner. Low-level Ideal Signatures and General Integrity Idealization. Information Security Conference (ISC 2004), LCNS 3225, pages 39–51, September 2004.

[6] Ran Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. 42nd Annual Symposium on Foundations of Computer Science (FOCS 2001), pages 136–145, October 2001.

[7] Veronique Cortier and Bogdan Warinschi. Computationally Sound, Automated Proofs for Security Protocols. 14th European Symposium on Programming (ESOP 2005), LNCS 3444, pages 157–171, April 2005.

[8] Danny Dolev and Andrew C. Yao. On the security of public key protocols. IEEE Transactions on Information Theory, IT-29(12), pages 198–208, March 1983.

[9] Joshua D. Guttman, F. Javier Thayer, and Lenore D. Zuck. The Faithfulness of Abstract Protocol Analysis: Message Authentication. 8th ACM Conference on Computer and Communications Security (CCS 2001), pages 186–195, November 2001.

[10] Peeter Laud. Handling Encryption in Analysis for Secure Information Flow. 12th European Symposium on Programming (ESOP 2003), LNCS 2618, pages 159–173, April 2003.

[11] Peeter Laud. Symmetric encryption in automatic analyses for confidentiality against active adversaries. 2004 IEEE Symposium on Security and Privacy, pages 71–85, May 2004.

[12] Romain Janvier, Yassine Lakhnech, and Laurent Mazaré. Completing the Picture: Soundness of Formal Encryption in the Presence of Active Adversaries. 14th European Symposium on Programming (ESOP 2005), LNCS 3444, pages 172–185, April 2005.

[13] Daniele Micciancio and Saurabh Panjwani. Adaptive Security of Symbolic Encryption. 2nd Theory of Cryptography Conference (TCC 2005), LNCS 3378, pages 169–187, February 2005.

[14] Daniele Micciancio and Bogdan Warinschi. Completeness Theorems for the Abadi-Rogaway Logic of Encrypted Expressions. Workshop in Issues in the Theory of Security (WITS 2002), January 2002.

[15] Andrew C. Yao. Theory and Applications of Trapdoor Functions (extended abstract). 23rd Annual Symposium on Foundations of Computer Science, pages 80–91, November 1982.