Digital Signatures with Encryption: Fact and Fiction

Extended Abstract

Tomasz M. Kozlowski and Scott A. Smolka
Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794-4400 USA
{tomek,sas}@cs.sunysb.edu

Abstract

We analyze a security protocol that combines digital signatures with data encryption. The analysis technique we utilize is temporal logic model checking as provided by the *Concurrency Factory* specification and verification toolset [CLSS96]. We show that the protocol is secure, providing privacy and proof of authorship. We demonstrate, however, that a version of the protocol in which confirmation messages are used is susceptible to an attack. This is the case even if the encryption and signature operations are different, contrary to what is claimed in [Sch94]. Finally, we show how the protocol with confirmation messages can be fixed by including the identity of the sender in messages transmitted by the protocol.

1 Introduction

Public-key cryptography [DH76, RSA78], which uses a public key for message encryption and a private key for message decryption, is a well-established method for the *private* transmission of sensitive data. Privacy in this case means that if two parties use public-key cryptography to exchange a message, an intermediary cannot intercept and read the message, intercept and modify the message, or fabricate a realistic-looking substitute message [Sch94].

In addition to privacy, users may also be interested in *proof of authorship*, such as that offered by handwritten signatures on paper documents. As pointed out in [Sch94], such proof must have the following characteristics: the signature must be unforgeable; the signature is authentic; the signature cannot be transferred from one document to another; the signature cannot be repudiated; and the signed document must be unalterable.

Digital signatures can be used for proof of authorship in a computer network, and certain public-key cryptosystems can be used to implement digital signatures. The basic idea is the following[1]. Alice uses her private key to *sign* a document; Alice sends the document to Bob; and Bob uses Alice's public key to *verify* the signature.

In this paper, we analyze a protocol that combines public-key cryptography with digital signatures, and therefore strives to offer both privacy and proof of authorship. The protocol appears

[1]As is traditional, we will use Alice and Bob as the protocol participants, sometimes abbreviated A and B.

in [Sch94, page 37] and we refer to it as DSE, for Digital Signatures with Encryption. Using temporal logic model checking as provided by the Concurrency Factory specification and verification toolset [CLSS96], we show that DSE is secure, providing privacy and proof of authorship.

[Sch94] also considers a version of DSE with the additional feature of *confirmation messages*: whenever someone receives a message, he or she sends it back to the sender as a confirmation of receipt. We refer to this extension of DSE as DSEC, and show that DSEC is susceptible to an attack that compromises privacy. This is the case even if the encryption and signature operations are *different*, contrary to what is claimed in [Sch94]. By "different" we mean here that the signing operation is not the inverse of the encryption operation even when the same key is used in both, and likewise for the signature verification and decryption operations. The attack in question was extracted using the counter-example facility of the Concurrency Factory's model checker.

Finally, we show how DSEC can be fixed by including the identity of the sender in the messages transmitted by the protocol. Again, model checking is used to verify privacy and proof of authorship.

The structure of the rest of the paper is as follows. Section 2 provides an overview of the Concurrency Factory's VPL system specification language and model checking facility. Section 3 presents the protocols we are considering along with the possible attack on the DSEC protocol. Section 4 describes how we modeled the protocols in VPL, and encoded their correctness properties in the modal mu-calculus, the temporal logic supported by the Concurrency Factory's model checker. The results of our verification runs are the topic of Section 5. Finally, Section 6 concludes.

2 The Concurrency Factory

The Concurrency Factory [CLSS96] is an integrated toolset for the specification, verification, and implementation of concurrent and distributed real-time systems such as communication protocols and process control systems. The main features of the Factory are: a *graphical user interface* that allows the non-expert to design and simulate concurrent systems using GCCS, a graphical process algebra; a *textual user interface* for VPL; a *suite of verification routines* that currently includes a local model checker for the modal mu-calculus [RS97], a very expressive temporal logic, and a local model checker for a real-time extension of the modal mu-calculus [SS95]; and a *graphical compiler* that transforms GCCS and VPL specifications into executable Facile [TLK96] code.

We briefly describe here the components of the Factory that are most germane to our modeling and analysis of DSE and its variants: VPL and the local model checker for the modal mu-calculus. VPL is well suited for specifying concurrent systems with non-trivial control structure and data structures, and which involve value passing (as opposed to pure synchronization). VPL-supported data structures include integers of limited size and arrays and records composed of such integers. Structural equivalence of data types is employed, i.e. two integer types are considered the same if their sizes are equal and two records are equivalent when they have fields of the same type appearing in the same order.

A system specification in VPL is a tree-like hierarchy of subsystems. A subsystem is either a *network* or a *process*. A network consists of a collection of subsystems running in parallel and communicating with each other through typed channels. Processes are at the leaves of the hierarchy. Each subsystem, be it a process or a network, consists of a header, local declaration and body. The header specifies a unique name for the subsystem and a list of "formal channels" (by analogy with formal parameters of procedures in programming languages). The names of the formal channels of a subsystem can be used in the body of the subsystem and represent events visible to an external observer.

Declarations local to a network include specifications of the subsystems of the network and channels for communication between subsystems, hidden from the outside world. The body of a network is a parallel composition of its subsystems. A subsystem declared within a network can be used arbitrarily many times; each time a new copy of the subsystem is instantiated with actual channels substituted for the formal ones. Actual channels must match the formal ones introduced in the header and must be declared either as local channels or formal channels of the network immediately containing the subsystem.

Declarations local to a process consist of variable and procedure declarations. Procedure bodies, like process bodies, are sequences of statements. Simple statements of VPL are assignments of arithmetic or boolean expressions to variables, and input/output operations on channels. Complex statements include sequential composition, `if-then-else`, `while-do`, and nondeterministic choice in the form of the `select` statement.

VPL programs can be verified using LMC, the Concurrency Factory's local model checker for the propositional modal mu-calculus. LMC first produces a graph representation of the logical formula under consideration and then computes the product of this graph with the labeled transition system (guaranteed to be finite-state) underlying the VPL program. This "product graph" is constructed on-the-fly: each product graph node has an associated boolean value (representing the value of a subformula in an LTS state) and the value of a node is computed based on the values of its successors. Values of the terminal nodes in the product graph can be decided immediately based on the node itself. Nodes are never constructed unless their values are needed to determine the value of a previously constructed node.

To further reduce the size of the product graph, LMC avoids interleaving process transitions whenever the order of interleaving cannot affect the truth of a logical formula. Intuitively, this is achieved by enlarging the granularity of process steps so as to avoid redundant interleavings. LMC is also equipped with a diagnostic facility which allows the user to request that the contents of the depth-first search stack be displayed whenever a certain "significant event" occurs—for instance, when the search first encounters a state at which a logical variable is determined to be either true or false. This technique can be used to great effect, for instance, to discover execution paths leading to deadlocked states or corresponding to potential attacks on security protocols.

3 The Protocols

The protocols we analyze aim to provide privacy and proof of authorship to communicating agents, that is to guarantee:

- *privacy* — whenever agent A signs a message and sends it to agent B, only agent B can read the contents of the message;
- *proof of authorship* — whenever agent B receives a message signed by agent A, agent A has indeed signed the message.

Note that for the proof of authorship property we do not require that A has both signed and sent the message that B has received since this cannot be guaranteed. B could forward the message it received from A to some other agent C after decrypting and encrypting it appropriately. C would thus receive a message that had been signed by A but had been sent to her by B and not A.

The protocols utilize digital signatures and encryption algorithms, both of which are public-key algorithms [DH76]. The private key of an agent is used to sign a message, and the public key is used to verify the message; i.e. to verify that the message has indeed been signed by the agent who claims to have signed it. Another unrelated pair of private and public keys is used for the decryption and encryption operations, respectively.

3.1 Digital Signatures with Encryption (DSE)

The basic protocol we verify is as follows [Sch94, page 37]:

1. Agent A signs a message with her private key.
   $S_A(M)$
2. Agent A encrypts the message and signature with agent B's public key and sends it to B.
   $E_B(S_A(M))$
3. Agent B decrypts the message with his private key.
   $D_B(E_B(S_A(M))) = S_A(M)$
4. Agent B verifies the message with A's public key and recovers the original message.
   $V_A(S_A(M)) = M$

We shall refer to this protocol as *Digital Signatures with Encryption* (DSE). Note that the choice of public key used for verification of the signature in step 4 depends on the plaintext sender ID associated with the message received by agent B. As will be shown, a modified version of this protocol, in which confirmation messages are used, can be attacked if this field is changed by an intruder.

3.2 DSE with Confirmation Messages (DSEC)

Consider a version of the protocol in which confirmation messages are used. Whenever an agent receives a message, he or she sends it back to the sender as a confirmation of receipt. This protocol, which we refer to as *DSE with Confirmation Messages* (DSEC), is given as follows [Sch94, page 38]:

1. Agent A signs a message with her private key, encrypts it with agent B's public key, and sends it to B.
   $E_B(S_A(M))$
2. Agent B decrypts the message with his private key and verifies the signature with agent A's public key, thereby verifying that A signed the message and, at the same time, recovering the message.
   $V_A(D_B(E_B(S_A(M)))) = V_A(S_A(M)) = M$
3. Agent B signs the message with his private key, encrypts it with agent A's public key, and sends it back to A.
   $E_A(S_B(M))$

4. Agent A decrypts the message with her private key, and verifies the signature with agent B's public key. If the resultant message is the same one A initially sent to B, agent A knows that B received the message accurately.
   $V_B(D_A(E_A(S_B(M)))) = V_B(S_B(M)) = M$

DSEC is vulnerable to an attack that compromises privacy. Assume that agent I is a legitimate system user with his own public and private keys. Further assume that I somehow gets access to a message that A has sent to B, and later resends this message to B, claiming that it comes from him (agent I):

1. Agent I sends the overheard message to agent B pretending to be the sender.
   $E_B(S_A(M))$

2. Agent B thinks that this is a legitimate message from agent I so he decrypts the message with his private key and then tries to verify agent I's signature with agent I's public key. The resultant message, which is pure gibberish, is:
   $V_I(D_B(E_B(S_A(M)))) = V_I(S_A(M))$

3. Agent B nonetheless follows the protocol and sends agent I a receipt.
   $E_I(S_B(V_I(S_A(M))))$

4. Agent I can decrypt the message with his private key, verify it with agent B's public key, sign it with his private key, and verify it with agent A's public key to obtain M. The protocol has thus failed. $V_A(S_I(V_B(D_I(E_I(S_B(V_I(S_A(M)))))))) = M$

Note that I was able to successfully attack the protocol because he could change the plaintext sender ID of the message A originally sent to B. By altering that field, I was able to fool B into believing that the message came from him (agent I) rather than from A. Below, we present a modification to the DSEC protocol that prevents such attacks by including the sender ID in the encrypted part of transmitted messages.

Schneier [Sch94, page 39] contends that

> The ... attack works because the encrypting operation is the same as the signature-verifying operation, and the decryption operation is the same as the signature operation. If the protocol used even slightly different operation for encryption and digital signatures, this would be avoided.

The above attack, however, shows that this is not true and that privacy can be compromised even if all the operations used in the protocol are different. In fact, the attack we have exhibited is the same as the one presented in [Sch94, page 38] with the important exception that signature signing and verification operations have not been replaced by their decryption/encryption counterparts (e.g. the message A originally sent to B is $E_B(S_A(M))$, not $E_B(D_A(M))$ as in Schneier's attack). Moreover, we have in no place assumed that a signature operation is semantically equivalent to the corresponding decryption operation, nor that a signature-verifying operation is semantically equivalent to the corresponding encryption operation.

By preserving the presence of signature-related operations in the attack, we are able to show that the attack is possible, again, even if all the operations used in the protocol are different. Rather, it is the willingness of B to acknowledge the receipt of gibberish that creates the vulnerability to an attack and not the fact that the encryption operation might be the same as the signature-verifying operation, and the decryption operation might be the same as the signature operation.

3.3 DSEC$'$ — The Corrected DSEC Protocol

The attack on DSEC was successful because the intruder was able to alter the sender ID field of a message it overheard. The fix to the protocol involves adding the identity of the sender to both messages of DSEC; these message should now be $E_B(S_A(M, A))$ and $E_A(S_B(M, B))$, respectively. Apart from the sender ID that gets signed and encrypted, the messages in the modified protocol also include the usual plaintext sender ID field.

The new protocol, which we call DSEC$'$, can be stated as follows:

1. Agent A attaches her ID to a message, signs the augmented message with her private key, encrypts it with agent B's public key, and sends it to B.
   $E_B(S_A(M, A))$

2. Agent B decrypts the message with his private key and verifies the signature with agent A's public key, thereby verifying that A signed the message and, at the same time, recovering the message.
   $V_A(D_B(E_B(S_A(M, A)))) = V_A(S_A(M, A)) = (M, A)$

3. Agent B checks whether the encrypted sender ID agrees with the plaintext sender ID. If so, B attaches his ID to the message, signs the augmented message with his private key, encrypts it with agent A's public key, and sends it back to A.
   $E_A(S_B(M, B))$

4. Agent A decrypts the message with her private key, and verifies the signature with agent B's public key. If the resultant message is the same one A initially sent to B, agent A knows that B received the message accurately.
   $V_B(D_A(E_A(S_B(M, B)))) = V_B(S_B(M, B)) = (M, B)$

If B discovers that the two sender ID fields in step 3 do not match, he can either send a response back to the original sender (the one whose ID is signed and encrypted) using the appropriate encryption key, or not send any response at all. He should not, however, send a response back to the agent whose ID appears in the plaintext sender ID field as this might lead to an attack on the protocol.

4 Modeling the Protocols in the Concurrency Factory

We modeled the protocols of Section 3 in the Concurrency Factory using VPL. The agents taking part in the protocols are modeled as VPL processes. Moreover, the intruder is modeled as a process that can perform any attack that we would expect a real-world intruder to be able to perform. The intruder is able to [Low96]:

- take part in the protocol as a regular "honest" agent;
- overhear and/or intercept messages sent by other agents;

- decrypt messages encrypted with his own public key;
- introduce new messages;
- replay any messages he has seen, possibly changing any plaintext parts, even if he does not understand the contents of the encrypted message.

The intruder has his own private and public keys and can take part in normal runs of the protocol. Other agents may initiate runs of the protocol with him and he may initiate runs of the protocol with other agents. The actions of the intruder are unobservable by other agents when he tries to attack the protocol, but not when he is engaged in a normal run of the protocol.

The only capability of the intruder that we do not model and which the intruder might possess is the ability to store messages the intruder sees for use in future attacks. Also, we only model a single initiator, single responder, and single intruder. Finally, we assume that the encryption and digital signatures algorithms are *strong*, in that an intruder cannot deduce the plaintext from the ciphertext without knowing the key [Sch94].

To determine if the protocols are secure, i.e. satisfy the privacy and proof of authorship properties, we encoded these properties in the modal mu-calculus, and used the Concurrency Factory's local model checker for the modal mu-calculus [RS97].

Privacy means that only the intended receiver of a message should be able to read its contents. This property can be stated as a conjunction of the two following properties:

- **AG** (not `I_read_As_msg_to_B`)
- **AG** (`I_sent_As_msg_to_B` $\Rightarrow$ (not `B_reads_Is_msg_first`))

where **AG** is the CTL operator [CE81] meaning "in every state." The first formula expresses the fact that agent I should never read the contents of a message agent A has sent to agent B. The second formula states that whenever I passes A's message to B, B will be able to read that message.

The proof of authorship property cannot be directly translated into CTL since it refers to the past rather than to the future. For example, the simple formula

- **AG** (not `A_sent_msg_to_B` $\Rightarrow$ **AG** (not `B_read_As_msg`))

does not express the property we want and is just false. This property states that in every state if agent A has not sent a message to agent B, B will never reach a state in which he reads a message from A.

To express the property properly we define the following two formulas, where **W** is the weak until operator of CTL, **AX** is the "in all next states" operator of CTL, and C and D are some events:

- C before D = not D **W** C
- C between Ds = **AG**($D \Rightarrow$ **AX A** (not D **W** C))

The first formula, C before D, states that C should happen before D happens. The second formula, C between Ds, states that between any two D's there should be at least one C. Having these auxiliary formulas, and assuming that intruder I is an intermediary between A and B, we can now state the proof of authorship property as the conjunction of the following four formulas:

- `A_sent_msg_to_B` before `I_sent_As_msg_to_B`
- `A_sent_msg_to_B` between `I_sent_As_msg_to_B`
- `I_sent_As_msg_to_B` before `B_read_As_msg`
- `I_sent_As_msg_to_B` between `B_read_As_msgs`

Consider the first two formulas. They express the fact that before agent I can pass agent A's message along to agent B, A must have sent a message to B; and after I has passed A's message destined for B to B, A must send another message to B before I can pass another message from A to B. Together, these two formulas mean that whenever agent I passes agent A's message to agent B, A must have sent a message to B. From the model of the protocol we know that the signal `I_sent_As_msg_to_B` occurs only when I passes the unmodified message he just received from A to B. This implies that whenever I passes A's message along to B the message has indeed been sent by A.

Repeating this reasoning for the last two formulas we get that whenever B receives A's message, the message must have been sent by I. Combining this implication with the previous one we obtain that whenever B receives a message from A the message has been sent by A and therefore has been signed by A. Notice that A did not need to send the message directly to B; rather, A's message might have been relayed to B by some intermediary.

5 Verification Results

Using the Concurrency Factory's local model checker for the modal mu-calculus, we have obtained the following results:

- Verified that DSE satisfies the privacy and proof of authorship properties.
- Identified a counterexample (a possible attack) to the privacy property in DSEC.
- Verified that DSEC′ satisfies the privacy and proof of authorship properties.

Figure 1 contains the execution times for these runs (in seconds) along with the size of the state space generated by each protocol's VPL description. These results were obtained on a Sparc 20 with 96 MB of RAM.

Protocol		Proof of authorship	Privacy
DSE	state space size	1623	981
	execution time (sec)	40	25
DSEC	state space size	3663	2147
	execution time (sec)	520	205
DSEC′	state space size	1221	836
	execution time (sec)	73	47

Table 1: State space sizes and execution times for the verification runs.

6 Conclusions

We have analyzed a security protocol that provides privacy and proof of authorship through the use of encryption and digital signatures. Model checking in the Concurrency Factory has shown that, under a certain reasonable set of assumptions, the original protocol is secure while an extension of the protocol to include confirmation messages is vulnerable to a privacy attack. Model checking has also shown that the extended protocol can be rendered secure by including the sender ID in the protocol's encrypted messages.

As future work, we plan to extend the Concurrency Factory's VPL system specification language to directly support cryptographic primitives such as those found in the Spi calculus [AG97]. This should facilitate the encoding and analysis of more sophisticated security protocols.

References

[AG97] M. Abadi and A. D. Gordon. Reasoning about cryptographic protocols in the spi calculus. In Mazurkiewicz and Winkowski [MW97], pages 59–73.

[CE81] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In D. Kozen, editor, *Proceedings of the Workshop on Logic of Programs,* Yorktown Heights, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1981.

[CLSS96] R. Cleaveland, P. M. Lewis, S. A. Smolka, and O. Sokolsky. The Concurrency Factory: A development environment for concurrent systems. In R. Alur and T. A. Henzinger, editors, *Computer Aided Verification (CAV '96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 398–401, New Brunswick, New Jersey, July 1996. Springer-Verlag.

[DH76] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT–22(6):644–654, November 1976.

[Low96] G. Lowe. Breaking and fixing the Needham-Schroder public-key protocol using FDR. In T. Margaria and B. Steffen, editors, *Proceedings of the Second International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '96),* Vol. 1055 of *Lecture Notes in Computer Science*, pages 147–166, April 1996.

[MW97] A. Mazurkiewicz and J. Winkowski, editors. *Proceedings of the Eighth International Conference on Concurrency Theory (CONCUR '97)*, volume 1243 of *Lecture Notes in Computer Science*, Warsaw, Poland, July 1997. Springer-Verlag.

[RS97] Y. S. Ramakrishna and S. A. Smolka. Partial-order reduction in the weak modal mu-calculus. In Mazurkiewicz and Winkowski [MW97].

[RSA78] R. L. Rivest, A. Shamir, and L. Aldeman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[Sch94] B. Schneier. *Applied Cryptography: Protocols, Algorithms and Source Code in C.* Wiley, 1994.

[SS95] O. Sokolsky and S. A. Smolka. Local model checking for real-time systems. In *Proceedings of the 7th International Conference on Computer-Aided Verification.* American Mathematical Society, 1995.

[TLK96] B. Thomsen, L. Leth, and T.-M. Kuo. A Facile tutorial. In *Proceedings of the Seventh International Conference on Concurrency Theory (CONCUR '96),* Vol. 1119 of *Lecture Notes in Computer Science*, pages 278–298. Springer-Verlag, 1996.