A FORMAL ANALYSIS OF A DIGITAL SIGNATURE ARCHITECTURE *

David Basin
ETH Zurich
basin@inf.ethz.ch

Kunihiko Miyazaki
Hitachi Systems Development Laboratory
kunihiko@sdl.hitachi.co.jp

Kazuo Takaragi
Hitachi Systems Development Laboratory
takara@sdl.hitachi.co.jp

Abstract We report on a case study in applying formal methods to model and validate an architecture for administrating digital signatures. We use a process-oriented modeling language to model a signature system implemented on top of a secure operating system. Afterwards, we use the Spin model checker to validate access control and integrity properties. We describe here our modeling approach and the benefits gained from our analysis.

Keywords: Formal methods, model checking, data integrity, security architectures.

1. Introduction

We report on a case study in modeling and validating an architecture for administrating digital signatures. The signature architecture is based on the secure operating system DARMA (Hitachi's Dependable Autonomous Realtime Manager), which is used to control the interaction between different subsystems, running on different operating platforms. In particular, DARMA is used to ensure data integrity by separating

*This research project is partially sponsored by Telecommunications Advancement Organization of Japan (TAO).

user API functions, which run on a potentially open system (e.g., connected to the Internet), from those that actually manipulate signature-relevant data, which run on a separate, protected system. The overall architecture should ensure data-integrity, even when the open system is compromised or attacked. We investigate the use of formal methods to validate that this is indeed the case.

More abstractly, we investigate how formal methods can be applied to model and validate the integrity and internal control of an industrial-scale information system. In our study, the architecture modeled is quite complex, involving multiple operating systems and processes communicating between them, which carry out security-critical tasks. A full-scale verification of a particular implementation would not only be impractical, the results would be too specialized. The key here is to find the right level of abstraction to create a model suitable for establishing the security properties of interest.

In our case, which is typical for many data-integrity problems, the relevant security properties concern restricting access to data (e.g., passwords and signatures), where a user's ability to carry out operations depends on past actions, for example, whether the user has been authenticated. Abstractly, these properties correspond to predicates on traces (i.e. sequences) of system events, which suggests building an event-oriented system model that focuses on processes, relevant aspects of their internal computation, and their communication.

Concretely, we model the signature architecture as a system of communicating processes, abstracting away the operational details of the different operating systems as well as functional details like the exact computations performed by different cryptographic primitives. The resulting model describes how processes can interact and semantically defines a set of event traces. We formalize security properties as (temporal) properties of these traces and verify them using the SPIN model checker [4].

Our application of model checking to validate data integrity properties of a security architecture is, to our knowledge, new. Of course, model checking is the standard technique used to verify control-oriented systems [3, 9] and is widely used in hardware and protocol verification. Our work shares with security protocol verification approaches like [10–11] an explicit model of an attacker, where attacker actions can be interleaved with those of honest agents. Our work is also related to the use of model checkers to validate software and architectural specifications [2, 6, 16] and it shares the same problem: the main challenge is to create good abstractions during modeling that help overcome the large, or infinite, state spaces associated with the model.

Organization. In Section 1.2 we present the signature architecture and its requirements. Afterwards, in Sections 1.3 and 1.4, we show how both the system and its requirements can be formalized and rigorously analyzed. Finally, in Section 1.5, we draw conclusions and consider directions for future work.

2. The Signature Architecture

2.1 Overview

The signature architecture is based on two ideas. The first is that of a *hysteresis signature* [14], which is a cryptographic approach designed to overcome the problem that for certain applications digital signatures should be valid for very long time periods. Hysteresis signatures address this problem by chaining signatures together in a way that the signature for each document signed depends on (hash values computed from) all previously signed documents. These chained signatures constitute a signature log and to forge even one signature in the log an attacker must forge (breaking the cryptographic functions behind) a chain of signatures.

The signature system must read the private keys of users from key stores, and read and update signature logs. Hence, the system's security relies on the confidentiality and integrity of this data. The second idea is to protect these using a secure operating platform. For this purpose, Hitachi's DARMA system [1] is used to separate the user's operating system (in practice, Windows) from a second operating system used to manage system data, e.g., Linux. This technology plays a role analogous to network firewalls, but here the two systems are protected by controlling how functions in one system can call functions in the other. In this way, one can precisely limit how users access the functions and data for hysteresis signatures that reside in the Linux operating system space.

Our model is based on Hitachi documentation, which describes the signature architecture using diagrams (like Figures 1 and 2) and natural language text, as well as discussions with Hitachi engineers.

2.2 Functional Units and Dataflow

The signature architecture is organized into five modules, whose high-level structure is depicted in Figure 1. The first module contains the three signature functions that execute in the user operating system space. We call this the "Windows-side module" to reflect the (likely) scenario that they are part of an API available to programs running under the Windows operating system. These functions are essentially

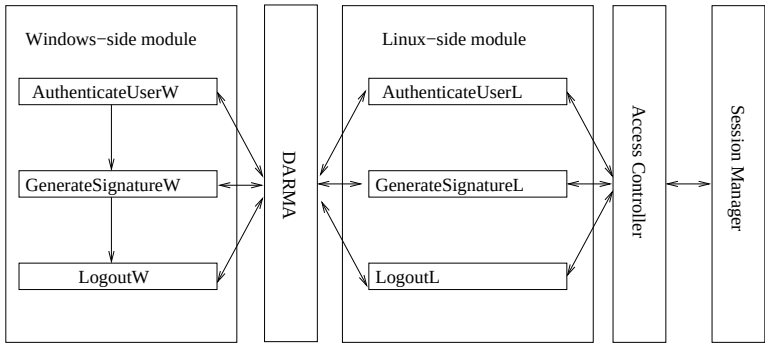

Figure 1. The Signature Architecture

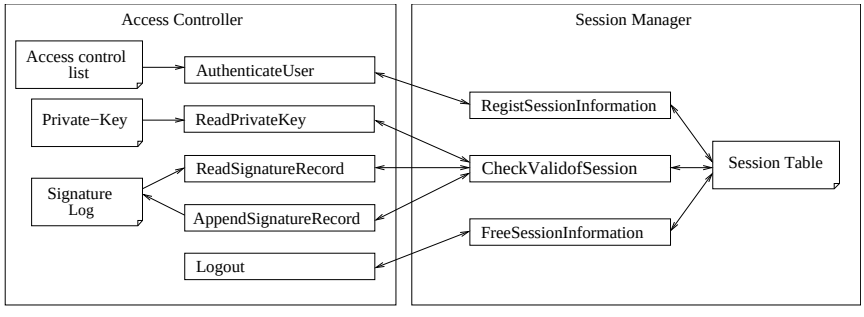

Figure 2. The Access Controller and Session Manager Modules

proxies. When called, they forward their parameters over the DARMA module to the corresponding function in the second, protected, operating system, which is here called the "Linux-side module", again reflecting a likely implementation. There are two additional modules, each also executing on the second (e.g., Linux) operating system, which package data and functions for managing access control and sessions.

To create a hysteresis signature, a user application takes the following steps on the Windows side:

1 The user application calls *AuthenticateUserW* to authenticate the user and is assigned a session identifier.

2 The application calls *GenerateSignatureW* to generate a hysteresis signature.

3 The application calls *LogoutW* to logout, ending the session.

Parameters
Input:
username: sent by *AuthenticateUserW* through *Darma*.
password: sent by *AuthenticateUserW* through *Darma*.
Output:
SessionID: If user authentication is successful, then $SessionID > 0$, otherwise $SessionID \leq 0$.

Details

1 Calculate hash value of *password* using Keymate/Crypto API. If successful, go to step 2, otherwise set *SessionID* to *CrypotErr* ($\leq$0) and return.

2 Authenticate user using the function *AuthenticateUser* of *Access Controller*.

3 Output *SessionID* returned by *AuthenticateUser*.

Figure 3. Interface Description for *AuthenticateUserL*

As explained above, each of these functions uses DARMA to call the corresponding function on the Linux side. DARMA restricts access from the Windows side to only these three functions. The Linux functions themselves may call any other Linux functions, including those of the *Access Controller*, which controls access to data (private keys, signature logs, and access control lists). The *Access Controller* in turn uses functions provided by the *Session Manager*, which manages session information (*SessionID*, etc.), as depicted in Figure 2.

The Hitachi documentation provides an interface description for each of these 16 functions. As a representative example, Figure 3 presents the description of *AuthenticateUserL*.

2.3 Requirements

The Hitachi documentation states three properties that the signature architecture should fulfill.

1 The signature architecture must authenticate a user before the user generates a hysteresis signature.

2 The signature architecture shall generate a hysteresis signature using the private key of an authenticated user.

3 The signature architecture must generate only one hysteresis signature per authentication.

In Section 1.4, we will show how to model properties like these in temporal logic.

3. Modeling the Signature Architecture

3.1 Process Modeling

Abstraction is the key to creating a formal model of the signature architecture. One possibility is to build a *data model* by formalizing the system data and the functions computed. Alternatively, we can focus on the dynamics of the system and build a *process* or *event-oriented model*.

We take the latter approach. One reason is that data and functions play a limited role in the system description. For example, the architecture description is abstract to the particulars of which cryptographic functions are used to hash or sign messages (i.e., those of the Keymate/Crypto API referred to in *AuthenticateUser* in Figure 3). A second reason is that the properties to be verified are event oriented and have a temporal flavor. They formalize that whenever certain events take place then other events have (or have not) also taken place. This suggests the use of temporal logic for formalizing properties and model checking for property verification.

There are two additional design decisions involved in creating our model, which are representative of the decisions arising in modeling any security architecture. First, we cannot completely abstract away data since control depends on data. In particular, the actions processes take depend on the values of keys, session identifiers, hash values, etc. The solution is to abstract large (or infinite) data domains into finite sets, and abstract functions over data to functions over the corresponding finite sets. The difficult part here is finding an abstraction that respects the properties of the functions acting on data. We will describe our approach to this in Section 1.3.3.

Second, to show that the security properties hold when the system is executed in a hostile environment, we must explicitly model the powers of an attacker. Here we adapt a common approach used in modeling security protocols [8, 12]: In addition to formalizing the system itself, we also formalize how different kinds of users can use the system. That is, we formalize (in Section 1.3.3) both normal "honest" users, who use the system as it is intended, and attackers who use the system in perhaps unintended ways and attempt to exploit and break into the system. The overall system model is built from submodels that define processes for each of the different subsystems together with the processes that model the normal users and the attacker. We then prove that the desired security properties hold of the system, even given all the possible malicious actions that can be taken by the attacker.

We have used the Spin model checker to formalize and check our model. Spin is a generic model checker that supports the design and

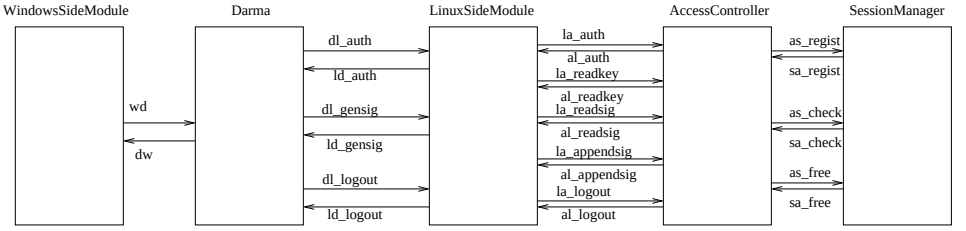

Figure 4. Modules and Channels

verification of distributed systems and algorithms. Spin's modeling language is called PROMELA (PROcess MEtaLAnguage), which provides a C-like notation for formalizing processes, enriched with process-algebra like primitives for expressing parallel composition, communication, and the like. Properties may be expressed in a linear-time temporal logic (LTL) and Spin implements algorithms for LTL model checking. Due to space restrictions, we will introduce Spin constructs as needed, on-the-fly. For a detailed description of Spin, the reader should consult [4–5].

3.2 Functions and Function Calls

As suggested by Figures 1 and 2, we can model the signature architecture in terms of five modules that communicate with each other in restricted ways. We will model each such module as a PROMELA process, where each process communicates with other processes over channels. A PROMELA channel is a buffer of some declared size that holds data of specified types. For each function in a module, we define two channels: one for modeling function calls and the other for modeling the return of computed values. This is depicted in Figure 4, which names the channels used for passing data between processes. All channels are declared to have size zero. According to the semantics of PROMELA, this means that communication on these channels is synchronous: the process sending data on a channel and the process receiving data from the channel must rendezvous, i.e., carry out their actions simultaneously.

As the figure shows, between Windows and DARMA we have just one calling channel *wd* and one returning channel *dw*.[1] This reflects that we have only one function in the *Darma* interface. This function is called

[1]Note that we ignore channels for calling the Windows functions since the functions that actually call *AuthenticateUserW*, *GenerateSignatureW*, and *LogoutW* fall outside the scope of our model, i.e., we do not consider who calls them, or how the caller uses the return values.

by marshaling (i.e., packaging) the function arguments together, including the name of the function to be called on the Linux side. We model this by putting all these arguments on the channel. For example, the expression *wd!AuthUser,username,password* (which occurs in our model of a normal user, given shortly), models that the function *AuthenticateUserW* calls *Darma*, instructing *Darma* to call *AuthenticateUserL* with the arguments *username* and *password*.

3.3 User Modeling

We now explain our formalization of the powers and actions of both ordinary users and system attackers.

The description of the signature architecture in Section 1.2 describes how the system is intended to be used by normal users. As we will see, it is a simple matter to translate this description into a user model.

The Hitachi documentation, describes, in part, the powers of an attacker, in particular that he cannot access functions on the Linux side. This is a starting point for our formalization of an attacker model, but it leaves many aspects open, for example, whether an attacker can operate "within" the system as a legitimate user with a valid password, or if he is an outsider, without these abilities. Moreover, it is unspecified what the attacker knows, or can guess or feasibly compute.

One achieves the strongest security guarantees by proving the safety of a system in the face of the most general and powerful attacker possible. Hence, we model an attacker who cannot only function as a legitimate user of the system, but can also call functions in unintended ways, with arbitrary parameters. Moreover, he knows, or can guess or compute, the names of other users, messages, and message hashes, and of course he knows his own password. However, we assume he can neither guess the passwords nor the session identifiers of other users. (If either of these were the case, then forging signatures would be trivial.)

We summarize these assumptions as follows:

1 The attacker can call *AuthenticateUserW*, *GenerateSignatureW*, and *LogoutW* in any order.

2 The attacker is also a legitimate user with a user name and a password.

3 The attacker knows the names of all users, and can guess messages and message hashes.

4 The attacker can only give his (good) password or a bad guessed password.

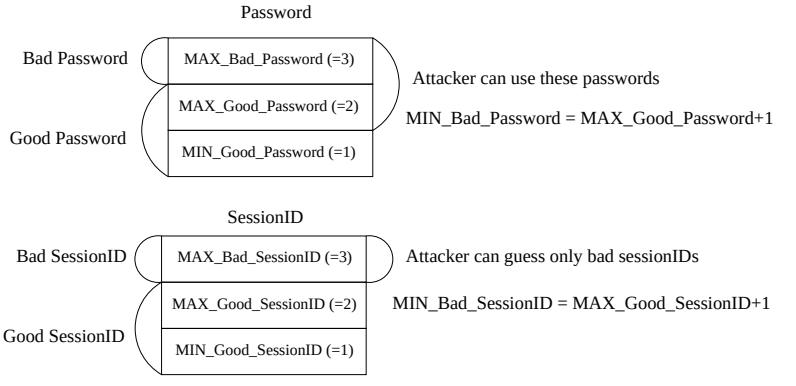

Figure 5. Modeling Passwords and Session Identifiers

5 The attacker cannot guess a good *SessionID*, i.e., one used by other users.

6 Generated *SessionID*s are always good.

In our model, we define sets of objects, namely finite intervals of natural numbers, for modeling the different kinds of objects in the problem domain: names, messages, hash values, and passwords. The key idea is to partition these sets into those things that are known by the attacker (or can be guessed or computed) and those that are not. For example, there is a set of user names, formalized by the set of natural numbers $\{MIN\_username, \ldots, MAX\_username\}$. We model that the spy knows, or can guess, any of these names by allowing him to guess any number in this set. However, we partition the ranges corresponding to passwords and session identifiers so that the attacker can only guess "bad" ones, which are ones that are never assigned to normal users. However, the attacker also has a "good" password, which allows him to use the system as a normal user and generate a good session identifier. Figure 5 depicts this partitioning, with the concrete values that we later use when model checking. For example, the good passwords are $\{1, 2\}$, where 2 represents the attacker's password. He can only guess passwords in the range {2,3}, where 3 models a bad password, i.e., one that does not belong to any normal user. As he cannot guess the password 1, he cannot use the system (e.g., to generate a signature) as any user other than himself.

Given this abstraction, it is now a simple matter to model the actions of normal users and the attacker.

```
1  proctype WindowsSideModule_Normal() {
2    byte username, password, sessionID, message, message_hash, signature, result;
3
4   setrandom(username, MIN_Good_Username, MAX_Good_Username);
5   setrandom(password, MIN_Good_Password, MAX_Good_Password);
6
7    do
8    :: wd!AuthUser,username,password;
9       dw?AuthUser,sessionID;
10
11     setrandom(message, MIN_Message, MAX_Message);
12     message_hash = Hash(message);
13
14     wd!GenSig,sessionID,message_hash;
15     dw?GenSig,signature;
16
17     wd!Logout,sessionID,0; /* second argument '0' is dummy */
18     dw?Logout,result
19  od}
```

Figure 6. User Model

Normal Users. Figure 6 shows our model[2] of a normal user, which directly models the steps that a normal user takes when using the signature architecture.

In lines 4 and 5 we model different possibilities for who uses the system and their messages. The macro *setrandom(x,lower,upper)* uses nondeterministic choice to set x to a value, $lower \leq x \leq upper$. Hence these lines set the username and password to those of a normal user, chosen nondeterministically from the predefined ranges.

Afterwards, the user generates a hysteresis signature. On line 8, the user calls *Darma* on the *wd* channel, specifying the execution of the Linux-side user authentication function, along with his username and password. On line 9, the result, a session identifier (whose value is greater than zero when authentication is successful), is returned on the *dw* channel.

On lines 11–12, a message from the space of possible messages is nondeterministically selected and its message hash is computed. We model *Hash* simply as the identity function. Although this does not reflect the functional requirements of a real hash function, in particular, that it is a one-way function, it is adequate for establishing the stipulated properties of our process model, which only rely on passwords and session

[2]Model excerpts are taken verbatim from our PROMELA model, with the exception of pretty printing, line numbering, and minor simplifications for expository purposes.

```
1 proctype WindowsSideModule_Attacker() {
2   byte username, password, sessionID, signature, dummy, result;
3   bit message_hash;
4
5   setrandom(username, MIN_username, MAX_username);
6   setrandom(message_hash, MIN_Message_Hash, MAX_Message_Hash);
7   setrandom(password, MAX_Good_Password, MAX_Bad_Password);
8   setrandom(sessionID, MIN_Bad_SessionID, MAX_Bad_SessionID);
9
10  do  /* Attacker calls these three functions in any order */
11  :: wd!AuthUser,username,password;
12     dw?AuthUser,sessionID
13
14  :: wd!GenSig,sessionID,message_hash;
15     dw?GenSig,signature
16
17  :: wd!Logout,sessionID,dummy;
18     dw?Logout,result
19
20     /* Or, Attacker guesses the following values */
21  :: setrandom(username, MIN_username, MAX_username)
22  :: setrandom(message_hash, MIN_Message_Hash, MAX_Message_Hash)
23  :: setrandom(password, MAX_Good_Password, MAX_Bad_Password)
24  :: setrandom(sessionID, MIN_Bad_SessionID, MAX_Bad_SessionID)
25  od}
```

Figure 7. Attacker Model

identifiers being unguessable. On line 14, the user calls *Darma* on the *wd* channel, instructing *Darma* to generate a signature with the session identifier returned from the previous round of authentication and a message hash. The generated signature is returned on line 15. Note that the return value can also indicate an error, e.g., if the session identifier was invalid.

Lines 17–18 model the user logging out, which invalidates his session identifier.

The Attacker. Figure 7 shows the PROMELA process that formalizes our attacker model. Here we see that the attacker can guess an arbitrary user name and message hash (lines 5–6). However, in accordance with the guessing model depicted in Figure 5, he can only guess one good password (*Max_Good_Password*), which allows him to log in as a normal user, or bad passwords (line 7). Similarly, he can only guess bad session identifiers (line 8).

Afterwards, we use a *do/od* loop with nondeterministic choice to model the attacker repeatedly calling *Darma* (on the *wd* channel) with

```
1   :: dl_auth?username_LINUX,password
2      ->  password_hash = Hash(password);
3         if
4         :: (password_hash <= 0) -> sessionID_LINUX = HashFunctionErr;
5                                    goto DONE_AuthL
6         :: else
7         fi;
8
9         la_auth!username_LINUX,password_hash;
10       al_auth?sessionID_LINUX;
11
12     DONE_AuthL:
13        ld_auth!sessionID_LINUX
```

Figure 8. AuthenticateUserL

these guessed values, in any order he likes. Alternatively, as modeled by the last four actions, he can guess new values at any point in time.

This example again illustrates the power of nondeterminism in a process-oriented modeling language. As with the user model, we use it to leave open which values are taken on by variables. This models a system where these variables can take on any value from the specified sets at system runtime. In addition, we use nondeterminism to describe the different possible actions that can be carried out by a user, while allowing the actions to be ordered in any way. The result is a succinct description of a general, powerful attacker. Of course, formalizations like this, which involve substantial nondeterminism, will typically lead to verification problems with large states spaces. But this can be seen as a feature, not a bug: model checkers can often search the resulting state spaces much quicker and more accurately than humans can.

3.4 Function Modeling

The majority of our PROMELA model describes the 16 functions contained in the system modules. As a representative function, we return to *AuthenticateUserL*, first described in Section 1.2.2.

Figure 8 shows the part of the PROMELA process that models *AuthenticateUserL* (this module also contains definitions for the other Linux-side functions). This directly models the three steps explained in Section 1.2.2: calculate a hash value (lines 2–7), authenticate the user (lines 9–10), and return the session identifier (line 13).

Here we have a simple example of how creating a rigorous specification forces us to make all definitions explicit. Step 1 of the textual explanation states "If [hash value calculation is] successful, go to Step 2 . . . ". But the Boolean predicate "successful" isn't defined. Such omis-

```
1 init {
2     run WindowsSideModule_Normal();
3     run WindowsSideModule_Attacker();
4     run Darma();
5     lsm = run LinuxSideModule();
6     run AccessController();
7     run SessionManager()}
```

Figure 9. Initialization Process

sions arise frequently. In this example, it is easy to determine what is intended by reading other parts of the specification. Here we formalize success by stating that a *HashFunctionErr* is generated when the password hash is less than or equal to zero, and the operation is successful otherwise. In general, not all ambiguities are so easily resolved. One of the benefits of using a formal specification language is that we are forced to be unambiguous at all times; PROMELA contains a syntax checker and automatically detects undefined symbols.

3.5 Putting It All Together

We build the overall model by composing in parallel the processes defined above. Namely, we compose the two processes formalizing the Windows-side module (as used by normal users and by the attacker) and the processes for the remaining modules. This is depicted in Figure 9. Note that we associate an identifier *lsm* with the process executing the Linux-side module. This will be used during verification to refer to particular labels in an invocation of the *LinuxSideModule* process, as described in the next section.

4. Verification

We now describe how we use Spin to show that our model has the intended properties. To do this, we formalize "bad" behavior (by formalizing and negating "good" behavior) as temporal logic formulae. Spin converts, on-the-fly, our PROMELA model of the system and the temporal logic formula to automata (reducing model checking to an automata-theoretic problem as described in [15]), and then constructs and searches the resulting product automaton. If Spin finds a trace accepted by this automaton, the trace explains how the system allows the bad behavior. Alternatively, if Spin succeeds in showing, by exhaustive analysis of the state space, that no errors exists, then the model is verified with respect to the property.

As an example, we formalize the first of the three properties described in Section 1.2.3. Our first requirement states that the signature architecture must authenticate a user before the user generates a signature. The bad property is therefore the negation of this. Informally:

The signature architecture generates a signature for an unauthenticated user.

To formalize this as a temporal property, observe that to generate a signature, we first require a valid session identifier, which is the result of a successful user authentication. Suppose that *UAS(uname,sID)* denotes that the user *uname* is authenticated with the session identifier *sID* and that *GHSS(sID)* represents that the signature architecture has generated a hysteresis signature with the session identifier *sID* (greater than zero). This can be formalized in PROMELA as follows.

```
#define UAS(uname,sID)   (LinuxSideModule[lsm]@DONE_AuthL
    && username_LINUX == uname  && sessionID_LINUX == sID)

#define GHSS(sID)     (LinuxSideModule[lsm]@DONE_GensigL
    && signature_LINUX > 0 && sessionID_LINUX == sID)
```

In these definitions, we reference labels (using `@`) in our PROMELA model to formalize that processes have reached certain points in their execution, and we use predicates on variables to express conditions on the system state.

We can now express the above informally stated property as

$$\exists s : \mathit{session}.\ \mathit{GHSS}(s)\ \ \mathit{before}\ \exists u : \mathit{user}.\ \mathit{UAS}(u, s)\,. \tag{1}$$

This is not yet a formula of linear-time temporal logic. First, "before" is not a standard LTL operator. However it can be expressed using the LTL operator "until", written as infix **U**, by defining A *before* B as $(\neg B)\ \mathbf{U}\ A$, i.e., A occurs before B if and only if $\neg B$ holds until A. In our case

$$\exists s : \mathit{session}.\ (\neg \exists u : \mathit{user}.\ \mathit{UAS}(u, s))\ \mathbf{U}\ \mathit{GHSS}(s)\,. \tag{2}$$

Second, we must eliminate the two quantifiers over sets. Since these sets are finite, we can replace each quantifier by finitely many disjunctions, i.e., the formula $\exists s : \mathit{session}.\, P(s)$ can be expanded to $P(s_1) \vee P(s_2) \cdots \vee P(s_n)$, where $s_1, \ldots, s_n$ are the finitely many model representatives of session identifiers.

The resulting property is automatically verified by Spin in 2 hours of computation time on a 450 MHz UltraSparc II workstation. In doing so,

it builds a product automaton with over 20 million states and searches over 70 million transitions. The formalization and verification of the other two properties is similar.

5. Discussion

It took approximately one man-month to build and analyze the signature architecture. This included considering alternative designs and studying different ways of specifying the requirements. The resulting model is 647 lines of PROMELA.

Although the formal analysis did not expose any design errors, the process itself was still quite valuable. During the formalization, we uncovered numerous ambiguities and omissions in the Hitachi documentation, such as missing cases and undefined values. For example, as described in Section 1.3.3, we needed to explicitly formalize implicit assumptions on the environment. Indeed, one reason why verification was successful is that, during the modeling process itself, we uncovered and fixed these oversights and omissions. Moreover, the discipline involved in creating the formal model improved our understanding of the design and helped us identify better solutions. This process of sharpening and improving a design is often one of the major benefits of using formal methods.

As a concrete result, we have produced a verified model. It serves as unambiguous documentation, with a well-defined mathematical semantics, for subsequent system development. It also provides a starting point for formally certifying the signature architecture with respect to standards like the Common Criteria.

As future work, we would like to explore the possibility of undertaking a full scale verification, perhaps as part of such a certification. An important step here would be to formally verify the correctness of the abstractions used, i.e., that the verification of our small finite model entails the verification of the corresponding infinite state system with unbounded numbers of interacting users and infinite data domains. Techniques based on data-independence, such as those of [7, 13], may help automate this task. It would also be interesting to supplement our process model with a data model that formalizes the properties of the signature architecture functions. Although this is not required for verifying the three requirements examined here, this would be necessary to go beyond treating cryptography as a "black box", e.g., to reason about the adequacy of different cryptographic mechanisms. This would also provide a more complete (formal) documentation of the design and provide a starting point for code verification.

References

[1] Toshiaki Arai, Tomoki Sekiguchi, Masahide Satoh, Taro Inoue, Tomoaki Nakamura, and Hideki Iwao. Darma: Using different OSs concurrently based on nanokernel technology. In *Proc. 59th-Annual Convention of Information Processing Society of Japan*, volume 1, pages 139–140. Information Processing Society of Japan, 1999. In Japanese.

[2] W. Chan, R. J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. D. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):498–520, July 1998.

[3] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press, 1999.

[4] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[5] G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

[6] Daniel Jackson and Kevin Sullivan. COM revisited: tool-assisted modelling of an architectural framework. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 149–158. ACM Press, 2000.

[7] Gavin Lowe. Towards a completeness result for model checking of security protocols. In *PCSFW: Proceedings of The 11th Computer Security Foundations Workshop*, pages 96–105. IEEE Computer Society Press, 1998.

[8] Gawin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proceedings of TACAS'96*, LNCS 1055, pages 147–166. Springer, Berlin, 1996.

[9] K.L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Publishers, 1993.

[10] C. Meadows. The NRL protocol analyzer: an overview. *Journal of Logic Programming*, 26(2):113–131, 1996.

[11] J. C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using Murphi. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 141–153, 1997.

[12] Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.

[13] A. W. Roscoe and Philippa J. Broadfoot. Proving security protocols with model checkers by data independence techniques. *Journal of Computer Security*, 7(1):147–190, 1999.

[14] Seiichi Susaki and Tsutomu Matsumoto. Alibi establishment for electronic signatures. *Information Processing Society of Japan*, 43(8):2381–2393, 2002. In Japanese.

[15] M.Y. Vardi and P. Wolper. Automata-theoretic techniques for modal logics of programs. *Journal of Computer and System Sciences*, 32:183–221, 1986.

[16] Jeannette Wing and Mandana Vaziri-Farahani. A case study in model checking software systems. *Science of Computer Programming*, 28:273–299, 1997.