Formal Specification and Verification of Control Software for Cryptographic Equipment

D. Richard Kuhn and James F. Dray

National Computer Systems Laboratory
National Institute of Standards and Technology
Gaithersburg, Md. 20899

6th Computer Security Applications Conference, 1990 (IEEE Computer Society Press)

1. Introduction

Microprocessor-based systems are increasingly being used to provide improved security. The improvements in security are often accomplished at the cost of increased complexity, as when a smart card microprocessor replaces a simple password system for network access control. Formal methods are recognized as an effective means of assuring the security of systems, and have been used in several military security applications over the past 15 years [Neumann *et al.*, 1974; Tagney *et al.*, 1977; Feiertag *et al.*, 1977; Neuman *et al.*, 1980; Young *et al.*, 1986; Levin *et al.*, 1989]. This paper reports on the application of formal methods to two civilian security-critical systems: the Token-Based Access Control System (TBACS), a ''smart token''[1] system that controls access to a network of workstations, and a message authentication device implementing the ANSI X9.9 message authentication standard [ANSI, 1986]. A state-based specification was prepared for the smart token system. The message authentication device specification used the Vienna Development Method, which is based on denotational semantics.

The projects were undertaken primarily as exercises in preparation for a larger project that is planned, but the results surpassed the initial goal of gaining familiarity with verification tools. It is noteworthy that no funding was available for formal methods work in either case. A verification tool, Unisys' Formal Development Methodology (FDM) [Eggert *et al.*, 1988], was obtained at no cost and the formal methods work was done as time permitted. Even with limited time available, we found the effort worthwhile. In the smart token access control system, several inconsistencies were found that led to improved security. In addition, a subtle error was discovered that could have compromised the security of TBACS, had it been released. A breakdown of hours and resources used in the access control system verification is given in section 2.8. The most interesting result of this work, beyond the increased assurance for TBACS security, is that it gives additional evidence that formal methods can be successfully applied to ''real world'' problems. Formal methods are rarely used today and are often rejected out-of-hand as being too difficult or expensive. Our experience has convinced us that, at least for small projects, or for small portions of large systems, formal methods are a practical and cost-effective adjunct to traditional software engineering methods.

2. The Token Based Access Control System

[1] Strictly speaking, a *smart token* is different from a *smart card*, although the two terms are often used interchangably. Both are hand-carried devices containing microprocessors and memory, but there is an ISO standard for smart cards. A smart token is typically larger than a smart card.

2.1. System Description

The Token Based Access Control System (TBACS) was developed as an experimental system to replace traditional password based systems. Based on the TBACS proof-of-concept, a Smart card based Access Control System (SACS) that incorporates the TBACS design and code is now under development. TBACS uses a portable device called a smart token to control access to the resources of networked computer systems. The TBACS smart token performs cryptographic authentication to identify the user and up to 100 computers which the user wishes to access.

The system configuration for TBACS consists of a number of workstations and host computers interconnected by a communications network. Each workstation on the network is connected to a reader/writer device, which provides the electrical interface between the TBACS token and the workstation. When the user inserts a token into the reader/writer, a program running on the workstation manages the authentication process by issuing a sequence of commands to the token and receiving the token's responses to these commands.

2.1.1. Hardware

The smart token consists of a plastic carrier containing a microprocessor and non-volatile memory. The carrier has the same major dimensions as a standard credit card, with six recessed metallic contacts along one edge. The reader/writer connects to the workstation through a standard asynchronous serial communications port, eliminating the need for a custom communications interface.

2.1.2. Software

The TBACS token responds to a set of 17 commands (see Table 1), which are implemented in firmware stored in the token's non-volatile memory. The firmware code is approximately 2,600 lines of C. The sequence in which these commands are executed is controlled by a set of flags which are checked at the first step of each command. If the flags are not set correctly, the given command will not be executed and the token will return an error code.

Table 1. TBACS Commands

Command	Verified
Reset	no
Enter SO PIN	yes
Authenticate SO	yes
Enter User PIN	yes
Load Key	yes
Authenticate Token	yes
Generate Challenge	yes
Authenticate User	yes
Change Token PIN	yes
Workstation Verify and Respond	yes
Output ID Table	no
Host Verify and Respond	yes
Read Zone	no
Write Zone	no
Append Zone	no
Call DES	no
Test	no

The commands are grouped into three general classes: security officer (SO) commands, user/workstation authentication commands, and user/remote host authentication commands. The SO commands provide for the initialization of new tokens by loading host IDs, cryptographic keys, and

PINs. The token is ready to be issued to the user after the SO has completed this initialization process. The remaining commands implement the authentications required by TBACS to control the login process.

2.2. Authentication Processes

For a user to gain access to computing resources on a network using TBACS, a series of authentications between the smart token, the user, and various host computers must be performed. TBACS selectively controls access to all computers on the network, including the user's local workstation. By taking advantage of the processing capabilities of the smart token, the login process can proceed transparently to the user while providing a high level of authentication. The DES algorithm, operating firmware, and critical data are stored internally on the smart token, providing a higher level of security than systems which use tokens only as data storage devices.

2.2.1. User/Token Authentications

When a user begins the login process on a workstation, he should have some means of determining the identity of the token. A program called the ''login manager'' is executed on the workstation when the user initiates a login, and is responsible for mediating the required series of authentications between the user, the token, and the workstation. First, the user must prove his or her identity to the token. The next step performed by the login manager is to request the token identification number from the token and display it on the user's screen for visual verification. The user can choose to either continue the login process or abort by simply pressing a key. The login manager prompts the user for his PIN/password, which is then encrypted and transmitted to the token along with the user ID. The token decrypts the user PIN and uses it as the key to encrypt the user ID. The result is then compared to the value stored on the token, and if these values match the token accepts the identity of the user. From this point on, TBACS uses the token to represent the user's identity for the remaining authentications.

2.2.2. Three-Way Handshake Protocol

Once the previous steps have been completed, the token and the workstation must authenticate to each other. This is accomplished through a three-way handshake protocol which allows each party to prove that it posesses the same cryptographic key as the other party, without having to physically exchange keys [NIST, 1988]. This protocol works as follows:

1 Party A generates a 64-bit random number and transmits it to party B.

2 Party B encrypts the random number using its secret key, generates a second random number, and transmits both values to party B.

3 Party A decrypts the first number and verifies the result. Party A then encrypts the second random number and transmits it to party B.

4 Party B decrypts and verifies the second random number. At this point, each party is satisfied that the other party posesses the same secret key.

2.2.3. User/Workstation Authentications

After the user and token authenticate to each other, the token must authenticate to the workstation. To perform the authentications between the workstation and the token, the login manager requests a random number from the token. The three-way handshake then proceeds with the token acting as party A and the workstation as party B. If this handshake is completed successfully, the login manager terminates and the user is logged in to the system.

2.2.4. User/Remote Host Authentications

At some point during a session, the user may decide to connect to a remote host via the network. The user activates an rlogin manager, which requests a table of the allowed TBACS hosts for this user from the token and displays this table in a menu format. After the user selects the desired remote host from this menu, the rlogin manager connects to an rlogin server on the remote host. At this point, the local rlogin manager acts primarily as a communications path between the token and the remote rlogin

server. The token is provided with the host ID, which it uses to select the proper key for subsequent cryptographic operations. The steps of the three-way handshake are repeated between the token and the rlogin server on the remote host, and finally the rlogin server terminates and the standard rlogin process connects the user to the remote host.

2.3. Token Deactivation

In addition to sequence control, the TBACS token is capable of deactivating itself after three failed login attempts or when the token expiration date is reached. Deactivation is accomplished by deleting the internal token identification number, after which none of the authentication steps required for user login will execute. A token is reactivated when a security officer installs a new token identification number.

2.4. Key Management

When a user first enrolls on a TBACS computer system, the user must contact the appropriate security officer for that computer. The SO initializes a blank token by loading the following: the security officer's ID, encrypted under the security officer's PIN; the user's ID, encrypted under an initial user PIN; a token identification number; and the token expiration date.

The SO next generates a DES key which is loaded onto the token. The random number generation capability of the security officer's token can be used to generate these keys. The token encrypts this key using the user's PIN and stores it in the key table along with the computer's host identification number. The host computer can generate this key from the user's PIN and the host master key as required during future login processes. As an alternative, the DES key could be stored in the computer's key database indexed by the user's identity. After receiving the token from the SO, the user may change the token identification number and the user PIN by entering the current values.

The user may now enroll on another TBACS computer by contacting that computer's SO, who generates another DES key which is stored on the token and the host computer as previously described. The TBACS token is designed so that only the SO who first initialized the token can delete token keys. Other security officers can only append keys to the token key table.[2]

In order to activate the token during a login, the user must supply the correct user PIN. Once activated, the token can be used to authenticate the user to the user's workstation and then to other host computers by means of the three-way handshake previously described.

2.5. Development

TBACS is a small but reasonably complex embedded system containing custom hardware. It was developed primarily as a proof-of-concept for the Smart card based Access Control System. Initially, a software simulation of TBACS was written to serve as a prototype. Experimentation with the prototype resulted in several design changes that were later incorporated into TBACS. The prototype also served as a specification for TBACS functions. Because of hardware requirements, most of the simulation code could not be used in the TBACS implementation. SACS, however, does incorporate almost all of the TBACS code. For this reason, the formal specification was based on the design as reflected in the TBACS code.

---

[2] Anyone can in fact append keys to the key table. This somewhat suprising feature was determined to be a reasonable design tradeoff. Security officers maintain control over the keys for their systems, and a user must have a valid key to access a particular host. A user can append a key, but it will be of no use unless it is the correct one that is controlled by the security officer. An alternative to this design would be to have each security officer store an encrypted secret key on the token, but this would require the token to be initialized by up to 100 security officers, since it is not known in advance which hosts a user will eventually need access to. Another alternative would be to have a ''master key'' that could be used by any security officer. But such a key would add little security, since a key known to over 100 people would likely be leaked in a short time in a civilian environment, where there are no criminal penalties for disclosure of confidential information.

The formal specification and verification were done after the TBACS hardware and software had been implemented because, as noted earlier, formal verification was not initially part of the development plan. Fortunately however, we were able to complete the verification before the implementation of the Smart card based Access Control System, allowing a problem detected in the formal verification to be corrected in the SACS implementation.

2.6. Security Policy

Generally accepted practice for developing trusted systems requires the statement of a security policy that describes the security properties of the system [NSA, 1985; Tavilla, 1986; Bell, 1988]. A formal model defining the meaning of the security policy in terms of mathematical logic can then be constructed. Confidence is gained in the security of the system by showing that it implements the requirements of the model. When a formal top-level specification of the system is prepared, its consistency with the model can be shown by rigorous mathematical argument. Proofs of lower level specifications and of the code may be formal or informal, depending on the complexity of the system and the resources available. Showing the consistency of the model with the policy statement is necessarily an informal process.

A formal model must be oriented toward a particular class of systems [Nessett, 1986]. For example, a model prepared for an operating system is not appropriate for expressing the security requirements for a network. Significant work has been done on the definition of formal models for multi-level secure operating systems [Bell and LaPadula, 1976; Feiertag *et al.*, 1977], and for trusted networks [Gove, 1985; Freeman *et al.*, 1988;]. Integrity models, such as those of Biba [1977], Lipner [1982], and Clark and Wilson [1987] are more directly related to TBACS verification requirements, but even these are not completely appropriate, so we developed a model that is particular to the requirements of TBACS.

Figure 1 summarizes the rules of operation that were originally defined as the security policy for TBACS, detailed in Dray *et al.* [1989] and Smid *et al.* [1989]. The original security policy was developed informally. The formal specification effort was started later. Initially we derived mathematical statements of the assertions given in Figure 1. However, it was not immediately clear that the conjunction of these assertions would guarantee the security of TBACS. For a greater degree of assurance, a more rigorously developed model of the security policy was required. The goal of this model development was to prepare a formal statement, P, of the security policy at a sufficiently abstract level that its security would be clear. Detailed assertions, $A_1, \ldots, A_n$, such as those in Figure 1, could then be stated and the model of TBACS functions shown correct with respect to these detailed assertions provided that $A_1 \,\&\, A_2 \,\&\, \ldots \,\&\, A_n \Rightarrow P$. This model and its derivation are documented in the next section.

2.7. Security Model

This section describes the derivation of the formal statement of security policy. In summary, TBACS security is defined as the conjunction of the following conditions:

1. *Access control:* Access to the network is granted only if the user posesses the correct PIN and a valid token. Ensuring this condition holds requires condition 2.
2. *Change control:* An invalid token cannot be made valid by the user, only by the security officer. Ensuring this condition holds requires condition 3.
3. *Privilege control:* A user cannot gain security officer privileges through manipulation of TBACS functions.[3]

[3] Note that this refers only to user actions within the system, and does not deal with actions that are beyond the control of TBACS, such as the user observing the security officer's PIN being entered, which is a separate concern.

1: Token commands may only be executed in legal sequences: user authorization; token authorization; workstation authorization; remote host authorization.

2: A token deactivates itself when its expiration date is reached.

3: A user must enter correct ID and PIN to be authorized:

4: A token deactivates itself after three failed login attempts

5: A deactivated token will not permit login.

6: A token allows a user access only to hosts whose ID and key are stored on the token.

7: The user cannot open the token if fail limit exceeded or token expired.

8: The user cannot get SO privileges.

9: An SO must enter correct ID and PIN to be authorized:

10: Only an SO may initialize a blank token.

11: A PIN for a particular token may be changed only by an SO or by the owner of that token.

12: After an SO has initialized a token, only this SO can enter the user PIN.

13: After an SO has initialized a token, only this SO can reactivate the token after it has been deactivated.

14: After an SO has initialized a token, only this SO can delete a key from the token.

15: Only the SO can change the expiration date.

Figure 1. Security Assertions

2.7.1. Terms

The primitive terms shown in Table 2 are used. In the remainder of the paper, the symbols &, ∣, ¬, => represent *and, or, not, implies,* respectively. The notation x' indicates the value of variable x after a state transition. The universal quantifier is denoted by **A** and the existential quantifier by **E**.

Table 2. Primitive Terms

exp_date	Token expiration date.
fail_log	Count of failed login attempts.
remote_host_authd	True iff remote host is authorized.
ws_authd	True iff workstation is authorized.
token_authd	True iff token is authorized.
user_authd	True iff user is authorized.
so_authd	True iff security officer is authorized.
today	Today's date.
user_pin	User PIN stored on token (encrypted).
ws_id	Workstation ID of the workstation to which token is connected.
host_ids	Host IDs to which the token may be used to gain access.
token_pin	Token PIN.
user_pin	User PIN.

2.7.2. Formal Statement of Model

2.7.2.1. Access Control

Access to the network is permitted only if the user possesses a PIN which encrypts the user ID to the value stored on the token, and the token is valid. That is,

(1) access => $E_{pin_in}(id_in)$ = user_pin & token_valid

where $E_K(I)$ represents the encryption of I with key K. Access is defined as authorization of remote host, workstation, token, or user. The token is valid when the token has not expired and is active, the failure limit has not been reached, and the workstation ID is in the token's host table. Substituting terms for these conditions into invariant (1) gives

(2) remote_host_authd | ws_authd | token_authd | user_authd
=> $E_{pin_in}(id_in)$ = user_pin &
today < exp_date &
fail_log < 3 &
token_pin ≠ null &
ws_id ∈ host_ids

2.7.2.2. Change Control

Invariant (2) must be maintained across state transitions. If the user could change the variables that determine if the token is valid, an invalid token could be made valid illegitimately. Thus for each variable in the definition of token_valid, we must define the conditions under which its value can change:[4]

The initial value for the user PIN can only be entered by the SO:
(user_pin = null & user_pin' ≠ null => so_authd)

A user PIN may be changed only by an SO or by the owner of that token:
(user_pin' ≠ user_pin => so_authd | user_authd)

Only the SO can change the expiration date:
(exp_date' ≠ exp_date => so_authd)

The failure log is reset only after a successful login.
(fail_log' = 0 & fail_log > 0 & fail_log < 3 => user_authd)

A PIN for a particular token may be changed only by an SO or by the owner of that token:
(token_pin' ≠ token_pin => so_authd | user_authd)

Only the SO can reactivate the token after it has been deactivated:
(token_pin = null & token_pin' ≠ null => so_authd)

2.7.2.3. Privilege Control

If it can be shown also that the user cannot obtain security officer privileges, the system is considered secure.

The user cannot get SO privileges:
user_authd => ¬ so_authd

2.7.3. Security Assertions

More detailed assertions were developed from the design description. Many of these are directly derivable from the formal policy statement given above. Others reflect design considerations, such as the sequencing requirement, assertion 1. The security model assertions are shown below. The FDM system notation has been replaced by standard first order logic notation to enhance readability.

State Assertions

These assertions must be true in all states.

[4] Recall that no restriction is placed on the addition of keys to the host table, as explained in Key Management, Section 2.4. Security in this case is external to TBACS and relies on the security officers for the different hosts maintaining confidentiality of keys.

1: Token commands may only be executed in the sequence: user authorization; token authorization; workstation authorization; remote host authorization:
(remote_host_authd => ws_authd) & (ws_authd => token_authd) & (token_authd => user_authd)

2: A user must enter correct ID and PIN to be authorized:
(user_authd => $E_{pin_in}(id_in)$ = user_pin)

3: A token deactivates itself when its expiration date is reached:
(today ≥ exp_date & ¬ so_authd => token_pin = null)

4: A token deactivates itself after three failed login attempts
(fail_log ≥ 3 => token_pin = null)

5: A deactivated token will not permit login:
(token_pin = null => ¬ user_authd)

6: A token allows a user access only to hosts whose ID and key are stored on the token:
(user_authd => ws_id ∈ host_ids)

7: The user cannot open the token if fail limit exceeded or token expired:
(user_authd => fail_log < 3 & today < exp_date)

8: The user cannot get SO privileges:
(user_authd => ¬ so_authd)

9: A security officer must enter correct ID and PIN to be authorized:
(so_authd => $E_{pin_in}(id_in)$ = so_pin)

10: Only an SO may initialize a blank token:
/* relies on external conditions, cannot be proven */

Transition Assertions

These assertions must be true of all transitions between states.

11: The initial value for the user PIN can only be entered by the SO:
(user_pin = null & user_pin' ≠ null => so_authd)

12: A PIN for a particular token may be changed only by an SO or by the owner of that token:
(token_pin' ≠ token_pin => so_authd | user_authd)

13: Only the SO can change the expiration date:
(exp_date' ≠ exp_date => so_authd)

14: A PIN for a particular token may be changed only by an SO or by the owner of that token:
(token_pin' ≠ token_pin => so_authd | user_authd)

15: The failure log is reset only after a successful login.
(fail_log' = 0 & fail_log > 0 & fail_log < 3 => user_authd)

16: Only the SO can reactivate the token after it has been deactivated:
(token_pin = null & token_pin' ≠ null => so_authd)

17: After an SO has initialized a token, only this SO can delete a key from the token:
((**E** h:host_id_t (¬ (h ∈ host_ids') & h ∈ host_ids)) => so_authd)

2.8. Verification

When the verification was started, TBACS had already been prototyped and built. The SACS system, based on TBACS, was undergoing critical design review during the verification effort. The verification thus served as an additional check on the soundness of the SACS design beyond the experimentation done with TBACS.

The security assertions and formal top level specification (FTLS) were specified using was Unisys' Formal Development Method (FDM) [Eggert *et al.*, 1988]. FDM provides a formal language

called Inajo [Scheid and Holtsberg, 1988] that is derived from first order logic. Inajo is used to represent the system as a state machine, specifying system states, state transitions, and security criteria. An interactive theorem proving tool [Schorre *et al.*, 1988] assists the user in showing that all transforms of the abstract machine meet the safety criteria. The ten critical functions that implement the three-way handshake and control functions were modeled with 297 lines (not including comments) of Inajo.

The verification resulted in some cases where the FTLS for a command could not be proven consistent with the security requirements. Such cases where the proof fails were the most interesting part of the verification. They prompted the discovery of some subtle discrepancies between the security requirements and the FTLS. With one exception, the discrepancies would not have compromised the security of the system, although they did expose design changes that could strengthen system security. The verification effort thus served to increase our confidence in the security of TBACS. This section discusses some of the anomalies that were discovered.

The security officer authentication procedure had no check that the input date parameter is greater than today's date, so assertion 2 could not be shown for this procedure:

token deactivates itself when its expiration date is reached:
(today ≥ exp_date => token_pin = null)

The SO could enter an expiration date that invalidated this requirement, although the user would still not be able to gain access in this case.

A more interesting condition occurred with the procedure for entering the user PIN. The following assertion could not be satisfied:

After an SO has initialized a token, only this SO can enter the user PIN:
(user_pin = null & user_pin' ≠ null => so_authd)

Proving this condition would have required assuming that no combination of key and data encrypted to null, i.e.:

A i:id, p:pin $E_p(i) \neq$ null)

which, although highly improbable, is not strictly valid.

This was because a user PIN initialized to 0 could match the encryption of the entered user ID with the user PIN if $E_{pin}(id) = 0$. The user PIN can be changed if either (the SO opened the token) or (the user opened the token and $E_{PIN}(ID)$ = stored encryption of user ID). If $E_{pin}(id) = 0$ and the stored encryption of user ID is 0, then the user PIN could be changed by user rather than just the SO.

Similarly, the system was designed to deactivate a token by setting its token PIN to null. Since the token PIN is an encrypted value, there was a small, but non-zero, probability that two values would be found that encrypt to null. In the implementation, special flags were used to indicate the conditions described above, rather than using null values.

The user authentication procedure allowed the failure counter to be bumped to 3 without clearing the token pin after it becomes equal to 3, so the assertion below could not be proven for this procedure.

A token deactivates itself after three failed login attempts:
(fail_log ≥ 3 => token_pin = null)

This occurred because the failure log was specified to be tested initially in the user authentication procedure but was not tested again after it was incremented when the user failed to authenticate. The token would of course be deactivated when the user attempted to log in again.

When the SO opened the token entering the procedure to change the token PIN, it was possible for the token to be expired but still active. The requirement that could not be satisfied is:

A token deactivates itself when its expiration date is reached:
(today ≥ exp_date => token_pin = null)

The initial design called for the token authentication to be performed before the user authentication. By the time the formal specification was prepared, this had been changed to perform the user authentication first. The security requirements thus had assertions that the user could not be authenticated if the token was expired or deactivated:

A deactivated token will not permit login:
(token_pin = null => ¬ user_authd)

user cannot open token if fail limit exceeded or token expired:
(user_authd => fail_log < 3 & today < exp_date)

However, the checks for token expiration and deactivation remained in the authenticate token procedure, so only the following assertions could be shown:

(token_pin = null => ¬ token_authd)
(token_authd => fail_log < 3 & today < exp_date)

The design was changed to do the validation in the user authentication procedure.

One error was discovered that could have compromised the security of TBACS. If a token deactivates itself because it expires or because there were too many unsuccessful login attempts, only the security officer should be able to reactivate it:

After an SO has initialized a token, only this SO can reactivate the token after it has been deactivated:
(token_pin = null & token_pin' ≠ null => so_authd)

An error in the token PIN change procedure allowed a user to reactivate a token that had been deactivated as a result of expiration. The token pin change procedure checked that either the user or security officer had been authenticated, without checking to see if the token was deactivated. In the original design this would have been acceptable, because the token was authenticated first, then the user. The token authentication procedure checked for a deactivated token and the sequence would have been stopped by this procedure. With the change to authenticate the user first, the token PIN change procedure could be invoked before the token had been authenticated.

2.9. Discussion

This section discusses the impact of the formal verification on the design and construction of TBACS.

2.9.1. Assurance/Confidence Gained

The most significant problem discovered was the possibility that a user could reactivate a token after it deactivated itself because the expiration date was reached or for other reasons. Preventing this error from being incorporated into a released product made the verification well worth the effort.

Although the other discrepencies detected in the verification would not compromise the security of the system, they indicated in some cases modifications that could be made to enhance the security of TBACS. The failure to prove the assertions *''A deactivated token will not permit login''* and *''The user cannot open token if fail limit exceeded or token expired''* is one such case. It is generally best to detect invalid login attempts as soon as possible. The protocol implemented by the system resulted in the date check and token deactivation check occurring in the token authentication function, rather than in the user authentication function, which occurs earlier in the authentication process.

2.9.2. Estimated Time Savings

Since the formal verification was performed after the design and implementation of TBACS was completed, it is difficult to estimate the time savings which might have resulted from applying these techniques through all phases of the project. The original software simulation of the TBACS command set consists of 2500 lines of C source code, and required approximately three man-months to complete. This code was used as a sounding board to test various ideas during the design of TBACS, and later served as the design specification for the system firmware.

Many design decisions were explored by modifying the simulation code and observing the results. In some cases, unexpected side effects were discovered. In other cases, some features proved to be impossible to implement. This approach provided a reasonable assurance that the system would perform as expected, but was labor intensive. By applying formal verification methods, the programming effort required to test ideas which proved to be inconsistent with the overall design might have been avoided. Since testing these ideas through code modification involved more than 50% of the programming effort, a significant time savings would have resulted. As shown in Table 3, the verification was approximately 6% of software design and development time.

Table 3. TBACS Development

Phase	Hours	Calendar Months
1. Design	320	2
2. Simulation	480	3
3. HW Development	751	2
4. SW Development	650	10
5. Acceptance test SW	140	1
6. Project Management	800	15
Verification	92	5
Verification/(1+2+4)	6%	

2.9.3. Limitations

Does the verification guarantee that TBACS and SACS are absolutely secure? No, the verification only checked the consistency of a formal specification, which is essentially a simplified *model* of TBACS, with the security criteria. Many code-level details are not dealt with by the formal specification. In addition, only the ten most critical functions were modeled. But the goal of the verification effort was only to verify the soundness of the design and detect errors that may not have been discovered in TBACS before the SACS system was completed. We believe this goal was achieved at a reasonable cost in time and resources. To our knowledge, TBACS is the first smart token application to have a formally verified design. The SACS design verification is being completed, representing the first application of formal verification to a smart card application.

The cost and difficulty of using formal methods rises very quickly with the depth to which verification is performed. A full code-level verification would probably be far too costly to justify on a system of this type. Other methods, such as code reviews and inspections, can be used to check the consistency of the code with the formal specification. Determining the optimal combination of formal verification and traditional methods is a challenging management problem.

3. X9.9 Message Authentication System

To support the need for secure electronic funds transfer (EFT) in both industry and its own bureaus, the U.S. Treasury Department initiated a program for certifying EFT equipment [Ferris, 1987; Treasury, 1986]. The EFT equipment being certified provides ANSI X9.9 Message Authentication capability [ANSI, 1986] and ANSI X9.17 Key Management functions [ANSI, 1984].

The equipment typically includes a secure microprocessor and a chip to perform encryption using the Data Encryption Standard [NBS, 1977]. Software controls access to the various functions through either password protection or magnetic cards. The software is usually small, approximately 4,000 lines of source code. Commercial developers supplying EFT equipment to the Treasury Department are required to develop it according to specifications given in [Treasury, 1986b]. The specifications mandate security features recommended in [NSA, 1986] and include requirements to aid in verification suggested by [NBS, 1982].

In an earlier project we had developed static analysis tools to assist in the certification of software used in EFT equipment. To improve the certification process, a proposal was made to prepare a formal reference specification of devices implementing ANSI X9.9. A preliminary draft of this specification was prepared using the Vienna Development Method (VDM) [Bjorner and Jones, 1978; Jones, 1980].

ANSI X9.9 defines the format of EFT messages, gives a digital signature algorithm using DES, and specifies several ways in which the signatures can be attached to messages. The digital signature, or Message Authentication Code (MAC), is computed by encrypting the first 64 bits of the message using DES [NBS, 1977] and then XORing the resulting 64 bits with the next 64 bits and feeding it back into DES. This continues until the end of the message. The leftmost 32 bits of the final output are used as the MAC.

A 64 bit key is used in computing the digital signature. Every key has a lifetime known as the cryptoperiod, which is defined as [ANSI, 1986]

> The time span during which a specific key is authorized for use or in which the keys for a given system may remain in effect.

To protect against message duplication each message comes with a message identifier which is unique for specific dates and cryptoperiod. The standard states that [ANSI, 1986]

> The message identifier, which shall be used as an authentication element, is a value that does not repeat before either the change of date or expiration of the cryptoperiod of the key used for authentication, whichever occurs first. I.e., there shall not be more than one message with the same date and the same message identifier that uses the same key.

Thus for all messages m_i, m_j, where $i \neq j$, the following invariant is maintained:

$$(\text{date}(m_i) \neq \text{date}(m_j) \mid \text{message_ID}(m_i) \neq \text{message_ID}(m_j) \mid \text{key}(m_i) \neq \text{key}(m_j))$$

Keys are distributed by a combination of manual and automatic means, and are indexed by a key ID. A message authentication device uses the key ID field contained in a message to determine which key should be used to authenticate the message. From a strict reading of the standard it was not clear whether a key ID can be reused after its cryptoperiod expires and whether or not a cryptoperiod can end in the middle of the day or only at midnight. If keys change only at midnight, then key IDs can be reused. If not, then some provision about key ID reuse must be specified to correctly specify the intent of using message identifiers to detect duplicate messages. Such provisions are not present in the standard and the phrase ''whichever occurs first'' implies that the cryptoperiod can expire at any time, because otherwise the expiration of the cryptoperiod could never occur before the change of date.

To understand the difficulty that a novice user might have in interpreting the standard, consider the following two messages. Remember that the key is not part of the message; the key ID is used to retrieve it from databases stored by both sender and receiver.

Date	Msg Id	Key Id	Key
Feb 2	23	9	AAAA
Feb 2	23	9	BBBB

Suppose that the cryptoperiod for key AAAA ends at 12:00 noon and that for key BBBB starts at 12:00 noon. If the message with key AAAA is sent at 11:59 a.m. and received at 12:01 p.m., which key should be used to authenticate the message? The cryptoperiod is defined as

The time span during which a specific key is authorized for use or in which the keys for a given system can remain in effect. In practice this is interpreted as requiring the key to be discontinued on both the sender's and receiver's ends at the same time.[5] Thus for our example, the message with key AAAA would fail to authenticate because the cryptoperiod for key AAAA had expired at noon, even though the message was sent before the expiration of the cryptoperiod. If the message with key BBBB were received after noon, it would authenticate correctly, regardless of when it was sent. Although this is the correct way to use X9.9, it may seem unsatisfactory to the user, raising questions about how the standard should be interpreted.

3.1. Discussion

The difficulty in interpretation occurred in the process of specifying a device to implement X9.9, rather than in verification. It has been suggested that the primary value of formal methods is in forcing a disciplined and thorough analysis during specification, rather than in the verification itself. The formal specification can be used to clarify aspects of the requirements that may not be clear from the natural language statement. This is valuable in software development for communication among different developers. In standards, the formal specification can be even more valuable because the standard is often used by hundreds or even thousands of developers who may have no communication with each other, yet who must produce software products that can work correctly with products that others produce according to the standard.

We found the Vienna Development Method very effective for specification. At a surface level, specifications in VDM are similar to Inajo (FDM) specifications, although the underlying theory is different. Anyone familiar with one method would have little trouble using the other. The extent to which VDM will be useful for verification is a question for future work. We hope to investigate this question in the future using the RAISE (Rigorous Approach to Industrial Software Engineering) tools developed by the European Esprit program [Neilsen *et al.*, 1989].

4. Conclusions and Future Directions

The formal specification and verification work described in this paper was undertaken primarily to gain experience with verification tools, such as FDM, in preparation for experimenting with the application of formal methods to safety-critical systems. The project surpassed this original goal, enabling us to contribute to the security of a critical system. Our experience with FDM was quite positive. We believe the system is easily accessible to anyone with a background in mathematical logic. As time and funding permit, we plan to acquire other verification tools as well.

Although formal methods are increasingly used on security critical systems, other application domains have seen relatively little use of these methods. We are interested in determining the extent to which the tools used in the work described in this paper can be applied to safety-critical systems, particularly medical devices. We hope to investigate the use of FDM, RAISE, and other tools in this domain in the near future.

5. References

ANSI [1984] Financial Institution Key Management (Wholesale), ANSI X9.17-1984, American National Standards Institute.

ANSI [1986] Financial Institution Message Authentication (Wholesale), ANSI X9.9-1986, American National Standards Institute.

Bell, D.E. [1988] ''Modeling Issue Paper,''National Computer Security Center, Contract No. MDA904-86-G-0028.

[5] This assumes that the cryptoperiod is the same for the sender and receiver, which is not actually required by the standard. The two messages in the example are in fact legitimate, and may occur in practice. Some organizations use a time stamp (which in effect gives a different key ID) to provide for messages that are sent within a key's cryptoperiod but received after its expiration.

Bell, D.E. and L.J. LaPadula [1976] ‘‘Secure Computer Systems: Unified Exposition and Multics Interpretation,’’ ESD-TR-306, Hanscom AFB, Bedford, Mass.

Biba, K.J. [April 1977] ‘‘Integrity Considerations for Secure Computer Systems,’’ Mitre Corp., TR-3153, Bedford, Mass.

Bjorner, D., and C.B. Jones [1978] *The Vienna Development Method: The Meta Language,* Lecture Notes in Computer Science, Vol. 61, Springer Verlag, New York.

Clark, D.D. and D.R. Wilson [1987] ‘‘A Comparison of Commercial and Military Computer Security Policies,’’ Proceedings, IEEE Symposium on Security and Privacy, Oakland, CA.

Dray, J.F., M.E. Smid, R.B.J. Warnar, ‘‘Implementing an Access Control System with Smart Token Technology,’’ *Proceedings, SCAT/ASIT Conference,* May 1989, Washington D.C., Miller-Freeman Publications.

Eggert, P., D. Cooper, S. Eckmann, J. Gingerich, S. Holtsberg, N. Kelem, R. Martin [December 1988] *FDM User Guide* Unisys Corp. TM8486/000/02.

Feiertag, R.J., K.N. Levitt, L. Robinson, [1977] ‘‘Proving Multilevel Security of a System Design,’’ *Proceedings, ACM Symposium on Operating Systems Principles.*

Freeman, J.W., R.B. Neely, G.W. Dinolt [1988] ‘‘An Internet Security Policy and Formal Model,’’ *Proceedings, 11th National Computer Security Conference.*

Ferris, M. and A. Cerulli, [1987] ‘‘Certification, A Risky Business,’’ *Proceedings, 10th National Computer Security Conference.*

Gove, R.A., [1985] ‘‘Modeling of Computer Networks,’’ *Proceedings, 8th National Computer Security Conference.*

Jones, C.B. [1980] *Software Development: A Rigorous Approach,* Prentice/Hall, Englewood Cliffs, N.J.

Levin, T.E., S.J. Padilla, R.R. Schell [1989] ‘‘Engineering Results from the A1 Formal Verification Process,’’ *Proceedings, 12th National Computer Security Conference.*

Lipner, S.B. [April 1982] ‘‘Non-Discretionary Controls for Commercial Applications,’’ Proceedings, IEEE Symposium on Security and Privacy, Oakland, CA.

McLean, J., [July, 1984] ‘‘A Formal Method for the Abstract Specification of Software’’, *Journal of the ACM*, Vol. 31, Nr. 3.

Neumann, P.G., [August 1974] ‘‘On the Design of a Provably Secure Operating System,’’ *Proceedings, International Workshop on Protection in Operating Systems.*

Neumann, P.G., R.S. Boyer, R.J. Feiertag, K.N. Levitt, L. Robinson [May 1980] ‘‘A Provably Secure Operating System: the System, Its Applications, and Proofs,’’ SRI Intl., Rpt. CSL-116.

NBS [1977] National Bureau of Standards, *Data Encryption Standard,* Federal Information Processing Standard, Publication 46, National Bureau of Standards, Gaithersburg, Md., 1977.

NBS [1982] National Bureau of Standards, ‘‘Software Validation, Verification, and Testing Technique and Tool Reference Guide,’’ *National Bureau of Standards Special Publication 500-93,* P.B. Powell, editor, National Bureau of Standards, Gaithersburg, Md., 1982.

Neilsen, M., K. Havelund, K.R. Wagner, and C. George [1989] ‘‘The RAISE Language, Method and Tools,’’ *Formal Aspects of Computing,* 1, 1, 1989.

Nesset, D. [1986] ‘‘Factors Afecting Distributed System Security,’’ *Proceedings, IEEE Symposium on Security and Privacy,* April, 1986, Oakland CA.

NIST [1988] Smart Card Technology: New Methods for Computer Access Control National Institute of standards and Technology, special Publication 500-157, National Technical Information Service, Springfield, VA, September 1988.

NSA [1985] National Security Agency, *Trusted Computer System Evaluation Criteria,’’* Dod 5200.28-STD.

NSA [1986] National Security Agency, X12, INFOSEC Standards and Evaluations Group, ‘‘Functional Security Requirements Specifications - NSA Specification 86-16’’. National Security Agency, Fort

Meade, Md.

Scheid, J., S. Holtsberg [Spetember 1988] *Inajo Specification Language Reference Manual*, Unisys Corp. TM-6021/001/04.

Schorre, D.V., D. Cooper, P. Eggert, J. Gingerich, G. Smith [November 1988] *The Interactive Theorem Prover (ITP) Reference Manual*, Unisys Corp. TM-6889/000/08.

Smid, M., J. Dray, R.B.J. Warnar, [1989] ''A Token Based Access Control System for Computer Networks,'' *Proceedings, 12th National Computer Security Conference.*

Tagney, J.D., S.R. Ames Jr., E.L. Burke, [June 1977] ''Security Evaluation Criteria for MME Message Service Selection,'' MTR-3433, MITRE Corp.

Tavilla, D.A. [June 1986] ''A Guide to Understanding the Orange Book Security Model Requirements,'' MITRE Corp.

Treasury [1986] U.S. Department of the Treasury Directive 16-02, ''Electronic Funds and Securities Transfer Policy -- Message Authentication and Enhanced Security,'' October 3, 1986.

Treasury [1986b] U.S. Department of the Treasury, ''Criteria and Procedures for Testing, Evaluating, and Certifying Message Authentication Devices for Federal E.F.T. Use'', Sept. 1, 1986.

Young, W.D., P.A. Telega, W.E. Boebert, R.Y. Kain [1986] ''A Verified Labeler for the Secure Ada Target,'' *Proceedings, 9th National Computer Security Conference.*