形式手法を用いたディジタル署名システムの安全性評価

宮崎 邦彦 David Basin 來間 啓伸 宝木 和夫 手塚 悟

本稿では，ディジタル署名システムのモデル化と検証に形式手法を用いたケーススタディについて述べる．われわれは，複数 OS(oprerating system) を一台の計算機上に共存させる DARMA と呼ばれるソフトウェア上に実装されたディジタル署名システムの安全性を検証した．具体的には，対象システムを PROMELA (PROcess MEta LAngage) を用いてモデル化し，さらに，SPIN モデルチェッカを用いてデータの完全性について検証を行った．以下では，われわれのモデル化のアプローチとその安全性検証から得られたメリットについて述べる．キーワード：モデルチェッキング，ディジタル署名システム，データ完全性

We report on a case study in applying formal methods to model and validate a digital signature system. We uses PROMELA (PROcess MEta LAnguage) to model the system implemented on top of DARMA which enable two different operating systems to work on the same computer simultaneously. Afterwards, we use the Spin model checker to validate integrity properties. We descrive here our modeling approach and the benefits gained from our analysis.

1 はじめに

本稿では，ディジタル署名システムのモデル化と検証に形式手法を適用したケーススタディについて報告する．対象としたディジタル署名システムは，株式会社日立製作所が開発した DARMA (Dependable Autonomous Realtime Manager) というソフトウェア上で動作する．DARMA とは，複数 OS(Operating System) を 1 台の計算機上で共存可能とするソフトウェアである．対象システムにおいて，DARMA は，署名に関係するデータを扱う空間を，利用者が直接扱う空間から隔離するために使われている．これにより，たとえ利用者が直接扱う空間側が，攻撃を受けたり，危殆化したりした場合であっても，データの完全性を保証することができる．われわれは，このアーキテクチャが実際にデータの完全性を保証できているかどうかを検証するために，形式手法を適用した．

本稿で対象とするシステムは，複数の OS とそれらの間の通信プロセスを含むかなり複雑な構成をしている．このため，システム全体を対象とした検証は現実的でない．また仮に検証できたとしても，その結果は特化しすぎたものになってしまう．したがって，対象とするセキュリティ要件の確認にふさわしいモデル化を行うための，適切なレベルの抽象化が重要となる．

本稿で検証対象とするセキュリティ要件は，多くのデータ完全性の問題と同様に，あるデータ (例：パスワード，ディジタル署名など) へのアクセス制御に関

A Formal Analysis of a Digital Signature System.

Kunihiko MIYAZAKI, Hironobu KURUMA, Kazuo TAKARAGI, Satoru TEZUKA (株) 日立製作所システム開発研究所

David BASIN, ETH Zurich

宮崎邦彦は，通信・放送機構（現：独立行政法人　情報通信研究機構）の委託研究「次世代証拠基盤技術に関する研究開発」として研究を行った．

コンピュータソフトウェア, Vol.22, No.2(2005), pp.74–84.

2004 年 7 月 16 日受付.

係している．このアクセス制御は，例えばユーザが認証されるなどのユーザの過去の動作に依存して実行される．すなわち，これらの要件は，システムイベントのトレースに関係している．したがって，モデル化に当たっては，イベント指向モデル，すなわちプロセスや関係する内部計算状態やそれらの間の通信に注目したモデル化が適切であると考えられる．

具体的には，対象システムを互いに通信しあうプロセスのシステムとしてモデル化し，複数の OS の詳細な動作や，署名生成のための暗号プリミティブの詳細な計算過程については，抽象化することでモデルから取り除いた．このようにして得られたモデルによって，プロセスがどう互いに影響するかが表現され，またイベントトレースが定義される．また検証対象となるセキュリティ要件を，これらのトレースの時相的性質として形式化し，SPIN モデルチェッカ [4] を用いて検証した．

著者らの知る限り，セキュリティアーキテクチャのデータ完全性を検証するために，モデルチェッキングを適用した例は他に見当たらない．モデルチェッキング自体は，制御システムの検証を行うための標準的な手法 [2] [10] であり，ハードウェアやプロトコルの検証に広く利用されている．本研究は，[11] [12] に見られるセキュリティプロトコル検証のアプローチと関連する部分がある．これらのアプローチにおいては，本研究の場合と同様に，正当なユーザとやり取り可能な攻撃者を明示的にモデル化して検証を行っている．また，[1] [6] [17] [9] [15] に見られるモデルチェッカやモデルファインダを用いたソフトウェアやアーキテクチャの検証に関する研究とも関連がある．これらの研究における課題は，本研究と同様に，モデル化に当たり，状態数の爆発を避けた適切なモデルをいかに構成するかという点にある．

以下，第 2 節において，対象とするディジタル署名システムの構成とセキュリティ要件について述べたあと，第 3, 4 節において，システムと要件それぞれが，どのように形式化され，厳密に評価されるかを説明する．最後に，第 5 節において結論を述べる．

2 対象システム

2.1 概要

対象とするシステムは，2 つの技術に基づいている．第 1 の技術は，ヒステリシス署名 [19] と呼ばれる技術であり，もうひとつは，DARMA と呼ばれる技術である．

ヒステリシス署名は，長期にわたり署名の有効性を保つことができるように設計されたディジタル署名技術である．ヒステリシス署名では，各電子文書に対する署名を生成する際に，あらかじめログとして保管しておいたそれ以前に生成された過去の署名データのハッシュ値に依存するように生成する．この結果，署名間には連鎖構造が構築される．この連鎖構造により，攻撃者にとっては，ひとつの署名を偽造するために，連鎖された複数の署名の偽造が必要となるため，長期経過後であっても (たとえば署名用秘密鍵が漏洩した後であっても)，偽造を著しく困難にできる．ヒステリシス署名に基づいて構成された対象システムにおいては，署名生成の際に，各ユーザの署名用秘密鍵の読み込みと，署名ログのデータの読み込み／更新が必要になる．このため，本システムのセキュリティは，これらのデータの秘匿性と完全性に依存している．

本システムが利用するもうひとつの技術である DARMA [20] [18] は，複数の OS を共存させる技術である．DARMA は資源管理機能と OS 間通信機能を提供する．資源管理機能は，メモリ，I/O 機器，プロセッサ等の計算機資源を分割し，各 OS および DARMA 自体に割り付ける機能である．メモリは指定された OS に占有となるため，ある OS 上のユーザプロセスが他 OS の領域にアクセスする危険性は非常に少ない．OS 間通信機能は，異なる OS 間での通信を可能とする機能であり，OS 間メッセージパシング，OS 間プロセス同期などの機能を提供する．これ以外の方法で，OS 間で通信を行うことはできない．またこの機能は OS から見るとある種のデバイスとして動作するため，ユーザプロセスから不正にアクセスすることは困難である．本システムでは DARMA を，利用者が直接操作を行う第 1 の OS (具体的には Microsoft(R)

Windows 2000[†1]) から，システムデータを管理する第 2 の OS (具体的には Linux[†2]) を隔離するために利用されている．この技術によって，第 2 の OS 側で管理されたヒステリシス署名関数や関連データへの，利用者からのアクセスを制限している．

以下，本システムの設計ドキュメントに示された対象システムのモジュール構成の概略と，セキュリティ要件を示す．

2.2 モジュール構成

対象システムは 5 つのモジュールから構成される (図 1)．1 番目のモジュールは，3 つの署名関数を持ち，第 1 の OS 上で実行される．以降，このモジュールを “Windows-side module” と呼ぶ．Windows-side module の関数は本質的に API のみを提供する．これらの関数が呼ばれたときは，DARMA を経由して，隔離された第 2 の OS 上で動作する対応する関数に引数が渡され，実際に実行される．以降，これら Windows-side module 側の関数に対応した関数を持つ第 2 の OS 上のモジュールを “Linux-side module” と呼ぶ．第 2 の OS 上ではさらに，アクセス制御を行う “Access controller” と，セッション管理を行う “Session manager” という 2 つのモジュールが動作する．

ヒステリシス署名を生成する際には，ユーザアプリケーションは第 1 の OS 側で以下のステップを実行する．

1. ユーザアプリケーションは，*AuthenticateUserW* を呼び，セッション ID を割り当てられる．
2. ユーザアプリケーションは，*GenerateSignatureW* を呼び，ヒステリシス署名を生成する．
3. ユーザアプリケーションは，ログアウトするために *LogoutW* を呼び，セッションを終了する．

上述したように，これらの Windows-side module の各関数は，DARMA を経由して，第 2 の OS 側の対応する関数を呼び出す．DARMA は，第 1 の OS 側からのアクセスを，これら 3 つの関数に制限する．第 2 の OS 側の関数は，さらに Access controller と Session manager という他の第 2 の OS 側のモジュールを呼び出すことがある．Access controller は，各種データ (秘密鍵，署名ログ，アクセスコントロールリスト) へのアクセスを制御する．また，Access controller は，各種セッション情報 (セッション ID など) を管理する Session manager を呼び出す．(図 2)

本システムの設計ドキュメントには，これらの各モジュールに含まれる合計 16 個の関数各々のインターフェイスが示されている．*AuthenticateUserL* のインターフェイス記述例を図 3 に示す．

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*.

図 3 Interface Description for *AuthenticateUserL*

2.3 セキュリティ要件

設計ドキュメントには，対象システムが満たすべき要件として，以下の 3 つが挙げられている．

1. ユーザがヒステリシス署名を生成する以前に，本システムはそのユーザを認証しなければならない．
2. 本システムは，認証されたユーザの秘密鍵を使ってヒステリシス署名を生成しなければならない．
3. 本システムは，ユーザ認証ごとに 1 回だけヒス

†1 Microsoft，Windows は，米国およびその他の国における米国 Microsoft Corp. の登録商標です．
†2 Linux は，Linus Torvalds の米国およびその他の国における登録商標あるいは商標です．

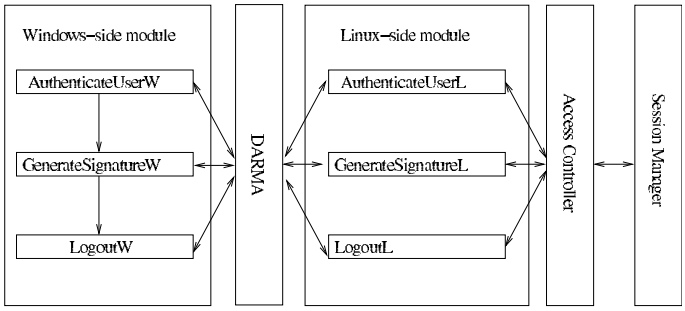

図 1 The Signature Architecture

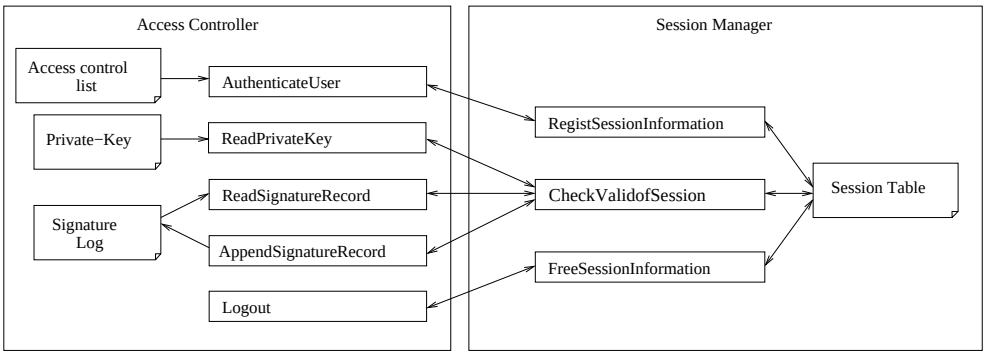

図 2 The Access Controller and Session Manager Modules

テリシス署名を生成しなければならない.

第 4 節において，これらの要件を時相論理でどのようにモデル化するかを説明する.

3 モデル化

3.1 プロセスのモデル化

対象システムの形式的なモデルを作るうえで，どのように抽象化するかが重要なポイントとなる．ひとつの可能性としては，対象システムが扱うデータと関数に着目して，**データモデル**を構築するというアプローチが考えられる．またもうひとつの可能性としては，対象システムの動作に着目し，**プロセス（イベント指向）モデル**を構築するというアプローチも考えられる.

本研究では，われわれは後者のアプローチを取ることとした．理由のひとつは，本システムの記述において，データや関数の役割が限定的であるためである．例えば，設計ドキュメントには，ハッシュ関数や署名関数などの本システムで利用される暗号関数は抽象的に書かれている (たとえば図 3 に示した *AuthenticateUser* 関数では Keymate/Crypto と呼ばれる暗号ライブラリの API を参照するにとどまっている)．またもうひとつの理由は，検証したいセキュリティ要件がイベント指向であり，時相的な性質を持つためである．これらの要件は,「あるイベントが起きたときはいつも他のイベントが起きる (起きない)」という形で形式化できる．このことは，これらの要件を時相論理として形式化し，モデルチェッキング手法によって検証できることを示唆している.

モデル化に当たって，さらに 2 つの課題がある．これらは，一般的なセキュリティアーキテクチャのモデル化においても共通しておこる課題である．1 つめの課題は，制御がデータに依存しているため，データを抽象化して完全にモデルから取り除くことはでき

ない，という点である．実際に，本システムにおける動作は，秘密鍵の値や，セッション ID，署名対象メッセージのハッシュ値などさまざまなデータに依存している．これに対する解決は，大きな (あるいは無限の) データ空間を有限集合として抽象化し，さらにデータ空間上の関数を有限集合上の関数として抽象化することである．ここで問題となるのは，元のデータ空間上の関数の性質を反映した適切な抽象化を見つけることである．この問題に対するわれわれのアプローチについては第 3.3 節で述べる．

2 つめの課題は，悪意のある環境下でシステムが実行された場合にセキュリティ要件を満たしているかどうかを検証するために，攻撃者の能力を明示的にモデル化する必要があるという点である．この点については，セキュリティプロトコルのモデル化 [8] [13] で行われているのと同様の一般的なアプローチをとった．すなわち，対象システムをモデル化するのに加え，対象システムを利用するさまざまなユーザについても形式化した．具体的には，第 3.3 節において，本システムを想定どおりに利用する正直な通常ユーザと，システムの不正利用や破壊を目的として想定外の方法で利用する攻撃者の 2 種類のユーザを形式化した．われわれが構築したモデルは，全体として，本システムを構成する各サブシステムに対応するプロセスと，通常ユーザと攻撃者それぞれに対応するプロセスとから構成される．このモデルの下で，攻撃者がどのような (悪意のある) 振舞いを行ったとしても，対象とするセキュリティ要件を本システムが満たしていることを検証する．

構築したモデルを検査するために，本稿では SPIN モデルチェッカを用いた．SPIN は，汎用的なモデルチェッカであり，分散システムやアルゴリズムの設計，検証を行うために利用される．SPIN の記述言語は PROMELA(PROcess MEtaLAnguage) と呼ばれる．PROMELA は，C 言語に似た記述方法をもち，並列処理の表現能力が高い．検証したい性質は，線形時間時相論理 (LTL) であらわすことができる．また，SPIN は on-the-fly でモデルチェックすることができる．すなわち，検証に必要な状態空間の全体を一度に構築することなく，効率的に検証可能である．SPIN についての詳細は，[4] [5] に述べられている．

3.2 関数と関数呼び出し

図 1,2 に示されるように，対象システムは，制限された方法で互いに通信しあう 5 つのモジュールによってモデル化できる．そこで，われわれは，これらのモジュールを PROMELA におけるプロセスとしてモデル化した．これらのプロセスは，PROMELA におけるチャネルによって接続されており通信を行う．PROMELA におけるチャネルとは，ある制限されたサイズをもつバッファーであり，特定の型のデータを蓄えるものである．われわれのモデルでは，モジュール内の関数毎に，2 つのチャネルを定義した．ひとつは関数呼び出しをモデル化したチャネルであり，もうひとつは戻り値をモデルしたチャネルである．図 4 に，本モデルにおいて定義した各プロセスやチャネルの名前を示す．すべてのチャネルのサイズは 0 と宣言されている．PROMELA の仕様により，これはチャネルが同期チャネルであることを意味する．すなわち，あるチャネルにデータを送信するプロセスと，そのチャネルからデータを受信するプロセスとで，それらの動作が同期的に実行される．

図 4 に示したように，第 1 の OS 側の Windows-side module と DARMA との間には，ひとつの呼び出しチャネル *wd* と戻りチャネル *dw* がある†3．これは，DARMA がただひとつの関数インターフェイスを持つことを反映している．この関数は，第 2 の OS 側の関数の名前や引数をひとまとめにして呼び出されるが，このことは，すべての引数をチャネル *wd* に送ることでモデル化する．たとえば，*AuthenticateUserW* 関数が DARMA を呼ぶことで，DARMA が第 2 の OS 側の *AuthenticateUserL* 関数を *username* と *password* を引数として呼ぶように指示する場合は，*wd!AuthUser,username,password* と記述することでモデル化される．

図 5 は，上記の方針にしたがってモデル化され

†3 Windows 側の各関数 *AuthenticateUserW*, *GenerateSignatureW*, *LogoutW* を呼び出す関数については検証対象外であるので，WindowsSideModule 側の各関数を呼び出すチャネルは本モデルには含めない．

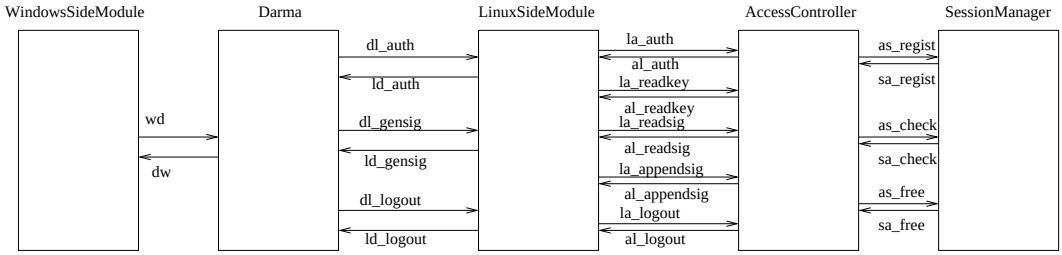

図 4 Modules and Channels

```
1   proctype Darma() {
2     byte username, password, sessionID, signature, dummy;
3     bit result, message_hash;
4
5     do
6       :: wd?AuthUser,username,password
7          ->
8             dl_auth!username,password;
9             ld_auth?sessionID;
10            dw!AuthUser,sessionID
11
12      :: wd?GenSig,sessionID,message_hash
13         ->
14            dl_gensig!sessionID,message_hash;
15            ld_gensig?signature;
16            dw!GenSig,signature
17
18      :: wd?Logout,sessionID,dummy
19         ->
20            dl_logout!sessionID;
21            ld_logout?result;
22            dw!Logout,result
23    od }
```

図 5 DARMA

た DARMA モジュールを示している．DARMA は，Windows-side module との間のただひとつの呼び出しチャネルである *wd* を使って送られてくるデータを受信し，その第 1 番目の要素が *AuthUser* であるか (6 行目)，*GenSig* であるか (12 行目)，*Logout* であるか (18 行目) に応じて，それぞれに対応する第 2 の OS 側の関数である *AuthenticateUserL* 関数，*GenerateSignatureL* 関数，*LogoutL* 関数を呼び出す．たとえば，*wd* からのデータの第 1 番目の要素が *AuthUser* であった場合には，DARMA は，*AuthenticateUserL* 関数の呼び出しチャネル *dl_auth* を使って *username,password* を送信し (8 行目)，戻りチャネル *ld_auth* から戻り値である *sessionID* を受信する (9 行目)．最後に，この *sessionID* を Windows-side module との間のただひとつの戻りチャネルである *dw* を使って windows-side module に戻す (10 行目)．

3.3 ユーザのモデル化

次に，通常ユーザと攻撃者の能力と振舞いの形式化について説明する．

通常ユーザが対象システムをどのように利用するかについては第 2 節に示した．後述するように，この記述をモデル化するのは容易である．

一方攻撃者の能力については，設計ドキュメントには，攻撃者は第 2 の OS 側の関数にはアクセスできない，など記述が一部見られるものの，多くの点については明記されていない．たとえば，攻撃者が有効なパスワードを知る正規のユーザとしてシステムを利用できるのか (すなわち内部不正者であるのか)，それともそうではない外部からの不正者を想定しているのか，といった点や，攻撃者が何を知っていて，何が予測可能であるのか，といった点は不明である．

できるだけ一般的で強力な攻撃者に対して対象システムが安全であることを示すことができれば，最も高いセキュリティを保証することができる．そこで，本検証では，攻撃者は，正規のユーザでもあり，第 1 の OS 側の各関数を (想定された正規の順番とは必ずしも一致しない) 任意の順番で，また任意の引数をつかって呼び出すことができるものとした．さらに，攻撃者は，他のユーザの名前，署名付与対象となるメッセージ，メッセージのハッシュ値，攻撃者自身のパスワードを知る (i.e. 内部不正者である)，または，予測できるものとした．ただし，攻撃者は他のユーザのパスワードやセッション ID については予測できないものとする．(これらの値を予測できる場合には，署名の偽造は容易である．)

以上の仮定をまとめると次のとおりである．

1. 攻撃者は，*AuthenticateUserW*, *GenerateSignatureW*, *LogoutW* を任意の順序で呼ぶことができる．

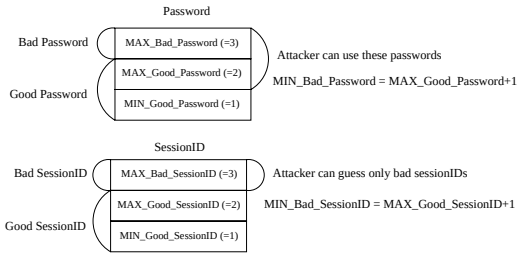

図 6 Modeling Passwords and Session Identifiers

2. 攻撃者は，正規のユーザでもあり，一組のユーザ名とパスワードを知っている．
3. 攻撃者は，すべてのユーザの名前を知っており，署名対象メッセージやメッセージのハッシュ値を推測できる．
4. 攻撃者は，自身の (正しい) パスワードと，推測した無効なパスワードを知っている．
5. 攻撃者は，正しいセッション ID，すなわち他のユーザによって利用されたセッション ID を予測できない．
6. システムによって生成されたセッション ID は，いつも正しい．

また，ユーザが取り扱う各種データ (ユーザ名，署名対象メッセージ，署名対象メッセージのハッシュ値，パスワード，セッション ID) をモデル化するために，これらのデータの集合を，自然数の有限区間として抽象化し定義する．さらに，これらの集合を攻撃者が知りうる (知っているまたは推測可能である) 範囲と，そうでない範囲に分割する．たとえば，ユーザ名の集合は，自然数の集合 $\{MIN\_username, \ldots, MAX\_username\}$ として形式化される．攻撃者はこれらすべての値を推測可能である．しかし，セッション ID については，攻撃者が推測できるのは正しくないもの，すなわち，正当なユーザには決して割り当てられないセッション ID だけである．ただし，攻撃者は正しいパスワードを知る正当なユーザでもあるので，そのパスワードをつかって通常ユーザとしてシステムを利用した場合には，正しいセッション ID が割り当てられる．図 6 は，パスワードとセッション ID それぞれについて攻撃者が知りうる範囲とそうでない範囲の分割の様子と，後述するモデルチェッキングにおいて実際に用いた具体的な値を示した図である．たとえば，正しいパスワードは $\{1, 2\}$ であり，このうち 2 は攻撃者のパスワードをあらわしている．また攻撃者が予測可能な範囲は $\{2, 3\}$ であり，3 は正しくない (つまりどの正当なユーザのものとも異なる) パスワードをあらわす．攻撃者は 1 を推測できないので，自身とは異なる他の正当なユーザとしてシステムを利用することはできない．

```
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}
```

図 7 User Model

以上の抽象化により，通常ユーザと攻撃者のモデル化が簡単になる．

通常ユーザ

図 7 は，上記の方針にしたがってモデル化された通常ユーザを示している．

4,5 行目はさまざまなユーザが利用する可能性をモデル化している．*setrandom(x,lower,upper)* というマクロは，$lower \leq x \leq upper$ を満たす x を非決定的に選択するために利用されている．したがって，4,5 行目で，ユーザ名とパスワードが決められた範囲から非決定的に設定される．

その後，設定されたユーザはヒステリシス署名を生成する．8 行目でユーザは DARMA を *wd* チャネルを使って，第 2 の OS 側でユーザ名とパスワードを使ったユーザ認証関数を実行するように指定して呼び出す．9 行目で，その結果であるセッション ID が *dw* チャネルを使って戻される (認証が成功したときには，戻り値が 0 より大きい)．

11,12 行目で，署名対象メッセージを決められた範囲から非決定的に選択し，そのハッシュ値を求める．

```
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}
```

図 8 Attacker Model

本モデルでは，*Hash* を単に恒等写像としてモデル化している．これは実際のハッシュ関数が持つ性質 (たとえば一方向性) を反映していないが，本モデルで検証する性質はパスワードとセッション ID が推測できないことだけに依存しているので十分である．

14 行目で，ユーザは DARMA を *wd* チャネルを使って，9 行目で受信したセッション ID と 12 行目で計算した署名対象メッセージのハッシュ値を引数として，第 2 の OS 側で署名生成関数を実行するように指定して呼び出す．セッション ID が不正などの理由でエラーが起きた場合には，戻り値によってその旨が示される．

17,18 行目でユーザはログアウトし，セッション ID を無効化する．

攻撃者

図 8 は，攻撃者をモデル化した PROMELA 記述を示している．攻撃者は，任意のユーザ名と署名対象メッセージのハッシュ値を推測可能である (5,6 行目)．しかし，図 6 に示したように，攻撃者は自身のもつひとつの正しいパスワード (*Max_Good_Password*) か，正しくないパスワードだけが推測可能である (7 行目)．同様に，攻撃者は正しくないセッション ID だけが推測可能である (8 行目)．

さらに，攻撃者がこれらの推測した値を引数として DARMA を任意の順序で呼び出す (11–18 行目)，あるいは，新しい値を推測する (21–24 行目)，ことを，*do/od* loop による非決定的な選択によってモデル化する．

```
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
```

図 9 *AuthenticateUserL*

この例は，プロセス指向モデリング言語における非決定性の能力を示している．攻撃者モデルにおいては，通常ユーザのモデルの場合と同様に，非決定性を使って，各変数に与えられる値を自由に取れるようにした．これは，実行時にこれらの変数の値を決められた集合の中から任意に設定されるシステムをモデル化したものである．さらに，任意の順序でユーザが各動作を実行できるということも，非決定性を使ってモデル化できた．これにより，一般的で強力な攻撃者が簡潔に記述可能となった．

なお，このような多くの非決定性を含む形式化は，しばしば大きな状態空間を要するという検証時の問題を引き起こす．しかし，これは欠陥と見るべきではなく，単に特徴と見るべきであると考えられる．実際，モデルチェッカは，人間より早く正確に状態空間をチェックできることが多い．

3.4 関数のモデル化

われわれの記述した PROMELA モデルの大半は，対象システムの各モジュールに含まれる 16 個の関数に関する記述である．具体的な例として，第 2.2 節に示した *AuthenticateUserL* 関数について説明する．

図 9 は，PROMELA モデルにおける *AuthenticateUserL* 部分を示している．これは，第 2.2 節に示した 3 つのステップをほぼ忠実にモデル化したものである．すなわち，まずハッシュ値を算出し (2–7 行目)，次にユーザを認証し (9,10 行目)，最後にセッション ID を返している (13 行目)．

ところで，ここに，厳密な仕様を作成するためにはすべてを明示的に定義しなければならないこと示す簡単な例を見ることができる．図 3 に示した自然言語に

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

図 10 Initialization Process

よる記述では，Step 1 は “If [hash value calculation is] successful, go to Step 2 . . . ” となっている．しかし述語 “successful” の定義は示されていない．このような省略はしばしば行われる．この例では，仕様の他の部分を参考にすれば，これが何を意図しているのかを決めることは難しくない．ここでは success というのを，もしパスワードハッシュが 0 以下であれば *HashFunctionErr* を出し，そうでなければ成功，とすることにより形式化した．しかし一般には，このような曖昧さを解決することはそう簡単でないこともある．形式的な仕様記述言語を利用することの利点の一つは，いつも曖昧さなしに記述するようになることである．

3.5 全体モデル

これまでに述べた各プロセスを次のように組み合わせて全体モデルを構築する．すなわち，それぞれ通常ユーザと攻撃者が使う Windows-side module を形式化した 2 つのプロセスと，他のモジュールをモデル化したプロセスを並列に組み合わせる (図 10)．ここで *lsm* は，次節で述べるように，検証の際に *LinuxSideModule* プロセスを参照するために使われる識別子である．

4 検証

本節では，前節までに構築したモデルが意図した要件を満たしていることを SPIN を用いてどのように検証するかについて述べる．検証を行うために，まず「悪い」振舞い，すなわち「よい」振舞いの否定，を線形時間時相論理式として形式化する．SPIN は，前節までに構築したシステムの PROMELA モデルとこの時相論理式をそれぞれオートマトンに変換し，それらの積オートマトンを構築し探索する (モデルチェッキングのオートマトン理論の問題への還元については[16] などに示されている)．もし SPIN がこのオートマトンによって受理されるトレースを見つけたとすると，そのトレースは，本システムが悪い振舞いをどのように許してしまうかを示すことになる．反対に，もし SPIN が全探索によってエラーが存在しないことを示せたとすると，このモデルが検査対象とした性質を満たしていることが検証されたことになる．

例として，第 2.3 節に示した 3 つの要件のうちの 1 番目の要件について，どのように形式化するかを説明する．第 1 の要件は，ユーザがヒステリシス署名を生成する以前に本システムはそのユーザを認証しなければならない，であった．したがって悪い振舞いとは，これの否定，すなわち,「本システムが，認証されていないユーザに対するヒステリシス署名を生成する」である．

これを形式化するためには，署名を生成する際には有効なセッション ID が必要であり，それはユーザ認証が成功したときに結果として与えられることに着目する．まず *UAS(uname,sID)* が，ユーザ *uname* がセッション ID *sID* によって認証されたことをあらわすものとし，また，*GHSS(sID)* が，ヒステリシス署名がセッション ID *sID*(ただし 0 より大きい) で生成されたことをあらわすものとする．これらは，次の PROMELA コードによって形式化される．

```
#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)
```

この定義において，プロセスがある特定の時点に達したことということを (記法@を使って) ラベルによってあらわしている．また，システムの状態に関する条件を表現するのに述語をつかっている．

これらの表記を使うと，上述の悪い振舞いは，次のように形式化される．

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

これはまだ線形時間時相論理 (LTL) になっていない．第 1 に “before” は，標準的な LTL の演算子ではない．この点については，LTL 演算子 “until”(**U**

と書く) をつかって，A *before* B を $(\neg B)$ **U** A と定義することで表現可能である．(i.e. A が B より前に (before) 起こるのは，A となるまでの間 (until)$\neg B$ であるとき，かつそのときに限る．)．すなわち，次のように書ける．

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

第 2 に，2 つの限量子を取り除く必要がある．これらの集合は有限であるから，限量子を有限個の論理和で書き換えることで取り除ける．すなわち，$\exists s : session.\, P(s)$ は，$P(s_1) \vee P(s_2) \cdots \vee P(s_n)$(ただし $s_1, \ldots, s_n$ は (有限個の) セッション ID をあらわす) と書き換えられる．

以上により得られた時相論理式は SPIN によって自動検証され，エラーは見つからなかった．この検証には 450 MHz UltraSparc II workstation 上で約 2 時間を要した．またこのときに，SPIN が構築した積オートマトンは 2,000 万以上の状態を持ち，探索範囲は 7,000 万以上の状態遷移に及んだ．なお，他の 2 つのセキュリティ要件についても同様に形式化，検証された．

5 まとめ

以上の検証作業にはおよそ 1 人月を要した．これには他のモデル化方針の検討や，要件を明確にするための別の方法の検討を含む．最終的に得られた PROMELA コードは 647 行となった．

これらの形式的な解析によって設計上のエラーは発見されなかったが，形式化する過程自体が有益であった．この間，条件分岐の不備や，未定義の値など，仕様上に多くの曖昧さや省略が発見された．また，たとえば第 3.3 節で述べたように，環境に関するさまざまな暗黙の仮定を，明示的に形式化する必要があった．実際，検証が成功した理由のひとつは，モデル化の過程で，これらの見落としや省略を見つけ，それを修正したためである．さらには，形式的なモデルを作ることは，設計に対する理解を深め，よりよい解を見つけるためにも役立つ．これは形式手法を用いることの最大の利点のひとつである．本研究と同様に，形式手法を設計段階において適用したときに得られる利点については，[3] などでも報告されている．

具体的な結果としては，われわれは検証可能なモデルを得ることができた．これは，曖昧さのないドキュメントを提供し，以降の開発に役立てられる．

今後の課題としては，システム全体を対象とした検証の可能性について探ることが考えられる．ここで重要なことは，本稿で利用した抽象化の正当性，すなわち，小さな有限のモデルによる検証結果が，これに対応する，任意のユーザ数と無限のデータ空間を持つ無限状態システムの検証結果を伴うかどうか，を形式的に検証することである．セキュリティプロトコルの評価においては，プロトコルで使われるユーザ名やノンスなどの具体的な値が，評価しようとする性質に影響を与えないことを利用して，小さな有限のモデルに対するモデルチェッキング結果から，無限状態を持ちうる実際のプロトコルの評価結果を導く data-independence という手法がある [7] [14]．この手法は，本稿で利用した抽象化，すなわちユーザ名，パスワード，セッション ID などを有限個に制限したこと，の正当性を示す上で適用できるかもしれない．実際に適用可能であるか確認することは今後の課題である．また，本稿で構築したプロセスモデルを，関数の性質を形式化したデータモデルで補完することもひとつの今後の方向性として挙げられる．今回対象とした 3 つの要件を検証する上では必要なかったが，ブラックボックスとしての扱いを超えて暗号関数を扱おうとする際には必要となる．またこれは完全な形式的な設計ドキュメントやコード検証を提供することにもつながる．

参 考 文 献

[1] Chan, W., Anderson, R. J., Beame, P., Burns, S., Modugno, F., Notkin, D., and Reese, J. D.: Model checking large software specifications, *IEEE Transactions on Software Engineering*, Vol. 24,No. 7(1998), pp. 498–520.

[2] Clarke, E., Grumberg, O., and Peled, D.: *Model Checking*, The MIT Press, 1999.

[3] Easterbrook, S., Lutz, R., Covington, R., Kelly, J., Ampo, Y., and Hamilton, D.: Experiences using lightweight formal methods for requirements modeling, *IEEE Transactions on Software Engineering*,

Vol. 24,No. 1(1998), pp. 4–14.
[4] Holzmann, G. J.: The Model Checker SPIN, *Software Engineering*, Vol. 23,No. 5(1997), pp. 279–295.
[5] Holzmann, G.: *Design and Validation of Computer Protocols*, Prentice-Hall, 1991.
[6] Jackson, D. and Sullivan, K.: COM revisited: tool-assisted modelling of an architectural framework, *ACM SIGSOFT Symposium on Foundations of Software Engineering*, ACM Press, 2000, pp. 149–158.
[7] Lowe, G.: Towards a Completeness Result for Model Checking of Security Protocols, *PCSFW: Proceedings of The 11th Computer Security Foundations Workshop*, IEEE Computer Society Press, 1998, pp. 96–105.
[8] Lowe, G.: Breaking and fixing the Needham-Schroeder Public-Key Protocol using FDR, *Proceedings of TACAS'96*, LNCS 1055, Springer, Berlin, 1996, pp. 147–166.
[9] Magee, J., Kramer, J., and Giannakopoulou, D.: Analysing the behaviour of distributed software architectures: a case study, *Proceedings of 6th IEEE Workshop on Future Trends of Distributed Computer Systems (FTDCS '97)*, IEEE Computer Society, 1997, pp. 240–245.
[10] McMillan, K.: *Symbolic Model Checking: An Approach to the State Explosion Problem*, Kluwer Academic Publishers, 1993.
[11] Meadows, C.: The NRL protocol analyzer: an overview, *Journal of Logic Programming*, Vol. 26,No. 2(1996), pp. 113–131.
[12] Mitchell, J. C., Mitchell, M., and Stern, U.: Automated analysis of cryptographic protocols using Murphi, *Proceedings of IEEE Symposium on Security and Privacy*, 1997, pp. 141–153.
[13] Paulson, L. C.: The inductive approach to verifying cryptographic protocols, *Journal of Computer Security*, Vol. 6(1998), pp. 85–128.
[14] Roscoe, A. W. and Broadfoot, P. J.: Proving Security Protocols with Model Checkers by Data Independence Techniques, *Journal of Computer Security*, Vol. 7,No. 1(1999), pp. 147–190.
[15] Sousa, J. P. and Garlan, D.: Formal modeling of the Enterprise JavaBeans^{TM}component integration framework, *FM'99 - Formal Methods, World Congress on Formal Methods in the Development of Computing Systems*, Lecture Notes in Computer Science, Vol. 1709, Springer, 1999, pp. 1281–1300.
[16] Vardi, M. and Wolper, P.: Automata-theoretic techniques for modal logics of programs, *Journal of Computer and System Sciences*, Vol. 32(1986), pp. 183–221.
[17] Wing, J. and Vaziri-Farahani, M.: A Case Study in Model Checking Software Systems, *Science of Computer Programming*, Vol. 28(1997), pp. 273–299.
[18] 佐藤雅英, 関口知紀, 新井利明, 井上太郎, 宮崎義弘, 梅都利和: ナノカーネル方式による異種 OS 共存技術「DARMA」の実装, **情報処理学会第 *59* 回全国大会**, Vol. 1, 情報処理学会, 1999, pp. 141–142.
[19] 洲崎誠一, 松本勉: 電子署名アリバイ実現機構 —ヒステリシス署名と履歴交差—, **情報処理学会論文誌**, Vol. 43,No. 8(2002), pp. 2381–2393.
[20] 新井利明, 関口知紀, 佐藤雅英, 井上太郎, 中村智明, 岩尾秀樹: ナノカーネル方式による異種 OS 共存技術「DARMA」の提案, **情報処理学会第 *59* 回全国大会**, Vol. 1, 情報処理学会, 1999, pp. 139–140.