平成21年度　修士論文

仮想化とヒステリシス署名を用いた
ログ改竄防止保全システムの実現

指導教員
齋藤 彰一 准教授

名古屋工業大学大学院 工学研究科
博士前期課程 情報工学専攻
平成20年度入学 20417569番

鶴田 浩史

平成22年 2月 4日

目 次

第 1 章　はじめに **1**

第 2 章　デジタルフォレンジックと関連研究 **3**

2.1 デジタルフォレンジック 3

2.1.1 デジタルフォレンジックの概要 3

2.1.2 デジタルフォレンジック技術の体系 4

2.2 関連研究 5

2.2.1 ログ保護手法 5

2.2.2 ログ保全手法 7

2.2.3 関連研究のまとめ 9

第 3 章　提案手法 **10**

3.1 概要 10

3.1.1 保全のための必要事項 10

3.1.2 保全システムの実現 11

3.2 全体構成 12

3.2.1 ヒステリシス署名 13

3.2.2 仮想化による保全機構 14

3.2.3 追記処理のみのログファイル 14

3.3 ログ保全処理 14

3.3.1 複数のログファイルの署名 14

3.3.2 高い頻度の署名 15

3.4 ログ保護処理 16

3.4.1 ログ追記処理 16
3.4.2 ログ削除処理 16
3.4.3 ログ改名処理 16

第 4 章 実装 18

4.1 複数のログファイルに対するヒステリシス署名 18
4.1.1 耐タンパデバイス 18
4.1.2 PKCS#11 19
4.1.3 ログファイルの自動分割 22
4.1.4 署名列 22
4.1.5 署名手順 24
4.2 署名検証 25
4.3 保全通信機構を用いた保護処理 27
4.3.1 Xen 27
4.3.2 通信機構 28
4.3.3 保全 OS の保全デーモン 30
4.3.4 ログファイルを開く処理 30
4.3.5 ログファイルを閉じる処理 33
4.3.6 ログ書き込み処理 34
4.3.7 ログファイルの削除処理 36
4.3.8 ログファイルの名前変更処理 38

第 5 章 評価・考察 42

5.1 評価 42
5.1.1 評価環境 42
5.1.2 書き込み速度 43
5.2 考察 43
5.2.1 署名間隔 43
5.2.2 改竄検出 44

第 6 章　まとめ **46**

謝辞 **47**

第1章

はじめに

インターネットの普及にともない，コンピュータを用いた犯罪による訴訟が増えている [1]．訴訟の原因として，ネット詐欺，誹謗中傷の書き込み，不正アクセス，個人情報などの機密情報の漏洩が挙げられる．訴訟に対して，コンピュータの操作記録やアクセス記録などのデジタルデータが証拠として扱われる．今後，操作記録やアクセス記録などのデジタルデータをログと呼ぶ．訴訟の際，ログを証拠として扱われることができるようにするには，ログを保全し，改竄や消去といった不正がされていないことを証明する必要がある．しかし，デジタルデータが記録されているファイルは加工が容易であるため，ログの記録されているファイルも同様に加工が容易となり不正の痕跡を改竄や消去されてしまい，証拠性が失われやすい．

そのため，法的な証拠性を保証するための技術であるデジタルフォレンジック [2] が重要視されている．デジタルフォレンジックは，コンピュータを用いた犯罪に対する訴訟が多くなってきていることにより注目され始めている技術である．そこで，デジタルフォレンジックの中でも，ネットワークに接続されていない環境下でログが記録されているファイルを保全するコンピュータフォレンジックに注目する．以後，ログの記録されているファイルをログファイルと呼ぶ．

ログファイルを保全するには，任意の場所を書き換えできないこと，root 権限の悪用による不正を防止できること，第三者がログファイルを検証できることの 3 つがすべて満たされる必要がある．

ログファイルの保護手法として，これまでに様々なシステム [3][4][5][6] が提案され

ている．これらのシステムの多くはログファイルを Write Once Read Many(WORM) デバイスに書き込むことでログの改竄を防止する．しかし，これらの手法では，システムからデバイスが外された場合にログファイルを改竄・削除される可能性がある．

また，ログファイルの保全手法として署名を用いるシステム [7][8] が提案されている．これらのシステムは，ログを署名することで改竄を検出できるため，改竄を困難にすることができる．しかし，システム管理者が root 権限を悪用することで不正な署名が行われる可能性がある．

そこで，本稿では，システム管理者が管理する OS とは別にログファイルを保全するために専用 OS を用いる手法と，ログファイルを保全するための OS で署名する保全手法を提案する．本手法を用いることにより，ログファイルに対する改竄や root 権限の悪用による不正を防止することができ，第三者がログファイルを検証することができるようになる．提案手法では，システム管理者による改竄の防止として仮想化を用いた追記のみのログファイルを用いる．また，ログファイルの改竄と消去の検出としてヒステリシス署名を用いる．

本稿では，2 章でデジタルフォレンジックとその関連研究について述べる．3 章で提案手法を述べ，4 章で実装について述べる．5 章で実験・評価を述べ，6 章でまとめと今後の課題を述べる．

第2章

デジタルフォレンジックと関連研究

本章では，法的な証拠性を保証するための技術であるデジタルフォレンジックについて述べ，ログの証拠性を保全する関連研究について述べる．

2.1　デジタルフォレンジック

ログの証拠性を保証するための技術であるデジタルフォレンジックの概要と技術体系を述べる．

2.1.1　デジタルフォレンジックの概要

コンピュータを用いた犯罪による訴訟に対して，裁判で使用するログファイルが重要になる．ログファイルは，電磁的記録であるため，加工が簡単である．そのため，ログファイルを証拠として扱うには，電磁的記録の加工の容易性が難点となる．そこで，ログファイルを裁判で使用できるようにするための技術がデジタルフォレンジックである．デジタルフォレンジックは，情報システム上で事件・事故が発生した際に，ログなどの電磁的な証拠を調べ，不正者・犯罪者の同定や事故原因・責任究明を行う技術 [9] である．

デジタルフォレンジック研究会 [10] によると，デジタルフォレンジックは，インシデント・レスポンス (コンピュータやネットワーク等の資源及び環境の不正使用，サー

ビス妨害行為，データの破壊，意図しない情報の開示等，並びにそれらへ至るための行為（事象）等への対応など) や法的紛争・訴訟に対し，電磁的記録の証拠保全及び調査・分析を行うとともに，電磁的記録の改竄・毀損等についての分析・情報収集等を行う一連の科学的調査手法・技術である，と定義している．

2.1.2　デジタルフォレンジック技術の体系

2.1.1 で述べたデジタルフォレンジックの定義から，デジタルフォレンジックは以下の 3 つの技術に分類することができる．

- コンピュータの操作記録やアクセス記録などのログやネットワークを流れるパケットを正確・確実に記録する技術
- 記録したログを保全する技術
- 記録したログからコンピュータの操作記録や収集したパケットから不正者・犯罪者や事故原因を分析・調査する技術

また，デジタルフォレンジックの体系としてネットワークフォレンジックとコンピュータフォレンジックに分けられる．コンピュータフォレンジックは，各コンピュータで，ログファイルの保全や原因解明に必要な証拠を特定する技術のことで，ネットワークフォレンジックは，ネットワーク上の通信パケットを収集し原因究明を特定する技術である．本稿では，コンピュータフォレンジックの中でも，ログファイルを保全する技術に注目する．

ネットワークフォレンジック

ネットワークフォレンジックは，ネットワークの利用状況や通信記録を収集して，ネットワーク障害の原因の分析や不正アクセスの行われた計算機の特定などを行う技術である．

コンピュータフォレンジック

コンピュータフォレンジックは，内部不正が行われた計算機から法的措置を取るための証拠を見つけ出すことや記録された情報を不正から保護する技術である．

2.2 関連研究

ログファイルの証拠性を保全するための研究が数多くされている．本節では，ログの改竄を防止する保護手法とログの改竄・消去を検出する保全手法の述べる．

2.2.1 ログ保護手法

ログファイルを外部からの攻撃から保護する手法 [3][4][6] は多い．ログファイルを保護する手法の特徴と問題点を述べる．

WORM への書き込み

ログファイルを WORM（Write Once Read Many）メディアに書き込むことで比較的容易にログファイルの改竄を不可能にできる．しかし，WORM メディアは，容量が限られているため，メディアのメンテナンスが難しい．そのため，WORM メディアの限られた容量を解決するための手法がある．

Wang らの手法 [4][5] は，ハードディスクを追記のみにすることで，改竄を防止するシステムである．Wang らの手法は，ログファイルを暗号化し，暗号化したログファイルと暗号化に用いた鍵を追記のみのデバイスドライバを通じてハードディスクに書き込むものである．ログファイルは，ハードディスクに書き込みを行う度に暗号化を行う．ログファイルを暗号化するために用いる鍵は，秘密鍵と公開鍵を用いる．秘密鍵は，耐タンパ領域に格納し，公開鍵を暗号化処理に用いる．ログファイルを暗号化するために，公開鍵とランダムな値から暗号化用の鍵を作成し，この鍵を用いてログファイルを暗号化する．そして，暗号化に用いた鍵を追記のみのデバイスドライバを通じてログファイルと共にハードディスクに書き込む．

また，追記のみのデバイスドライバは，ハードディスクにアクセスするためのブロッ

クデバイスを変更して実現している．ハードディスクを 2 つのパーティションに分割する．1 つのパーティションをセクター単位で書き込みを 1 回のみに制限する．書き込みを 1 回のみに制限したパーティションはすべてのビットを 0 に初期化し，書き込みを行ったセクターを 1 にセットすることでそのセクターに対する書き換えをできなくするものである．これにより，追記のみのデバイスドライバは，ハードディスクにアクセスするためのブロックデバイスを基にセクター単位で書き込みを制御しているため，ハードディスクに保存されているログファイルを改竄することは不可能である．

Debiez らの手法 [6] は，デバイスを追記のみに制御する API を用いることで，改竄を防止するシステムである．Debiez らの手法は，WORM モジュールと呼ばれる特別な API を使用して，ハードディスクへの書き込みに対する要求をフックし，WORM モジュールで書き込みに対する要求を追記にするものである．ユーザからの要求を WORM モジュールで制御することでログファイルを改竄することを不可能にできる．

しかし，これらの手法 [4][6] は，内部からの攻撃を防ぐことができない．ログファイルが保存されているハードディスクをシステムから外され，他の計算機でログファイルを改竄・消去される攻撃に対処することができない問題がある．また，このような攻撃をされた際，ログファイルを改竄されたことを検出することができない．

逃げログ

高田らの手法 [3] は，ログファイルのバックアップを作成することでログファイルの改竄を困難にする手法である．高田らの手法は，ログファイルをバックアップファイルを複数作成し，作成した複数のバックアップファイルをファイルシステム内の任意のディレクトリに隠蔽するものである．隠蔽するには，バックアップの数は任意で作成し，書き込み可能な任意のディレクトリにバックアップを作成する．

また，バックアップのファイル名はシステムが無造作に作成し，定期的に隠蔽処理をすることでログファイルの改竄を困難にすることができる．隠蔽処理は定期的に行われログファイルの原本とバックアップは，異なるディレクトリに任意のファイル名で隠蔽する．

しかし，攻撃者にバックアップを含めたすべてのログファイルを発見された場合，ロ

グファイルを改竄される可能性がある．また，定期的な隠蔽処理はユーザが指定するため，システムの安全性を保証しにくい．複数のバックアップを作成するため，システムの負荷がバックアップ数に比例して大きくなり，安全性が失われる．

2.2.2 ログ保全手法

ログファイルに対して何らかの改変が行われたとしても第三者が改変を検出する保全手法が提案されている．保全手法の特徴と問題点を述べる．

MAC を用いた署名

Schneier らの手法 [8] は，ログファイルの MAC(Message Authentication Code) を生成する際に用いる署名鍵を処理ごとに変更するものである．MAC は，ログファイルが改竄されていないことを確認するために使うメッセージダイジェストのことで，MAC を生成する際に，署名鍵を処理ごとに更新と破棄を行うことで，ログファイルの改竄を不可能にする．

しかし，検証者が最初の署名鍵を持っているため，検証者がログファイルを改竄することが可能になる．従って，検証者がログファイルを改竄して署名しても後からそれが改竄されていることを検出することができない問題があるため，ログファイルを保全するには適さない．

ヒステリシス署名を用いたデジタルフォレンジック

芦野らの手法 [7] は，セキュリティデバイスとヒステリシス署名 [11][12] を用いることでログファイルを保全する手法である．芦野らの手法は，耐タンパ領域を備えたセキュリティデバイスにヒステリシス署名をした際の最新の署名を保管することで，ログファイルの改竄を不可能にする．

しかし，ログを取得する機能と取得したログを署名する機能がユーザ空間に存在するため，それらの機能が改竄され，ログファイルが改竄される可能性がある．従って，システムを提供する管理者がログファイルを改竄することも可能になる．

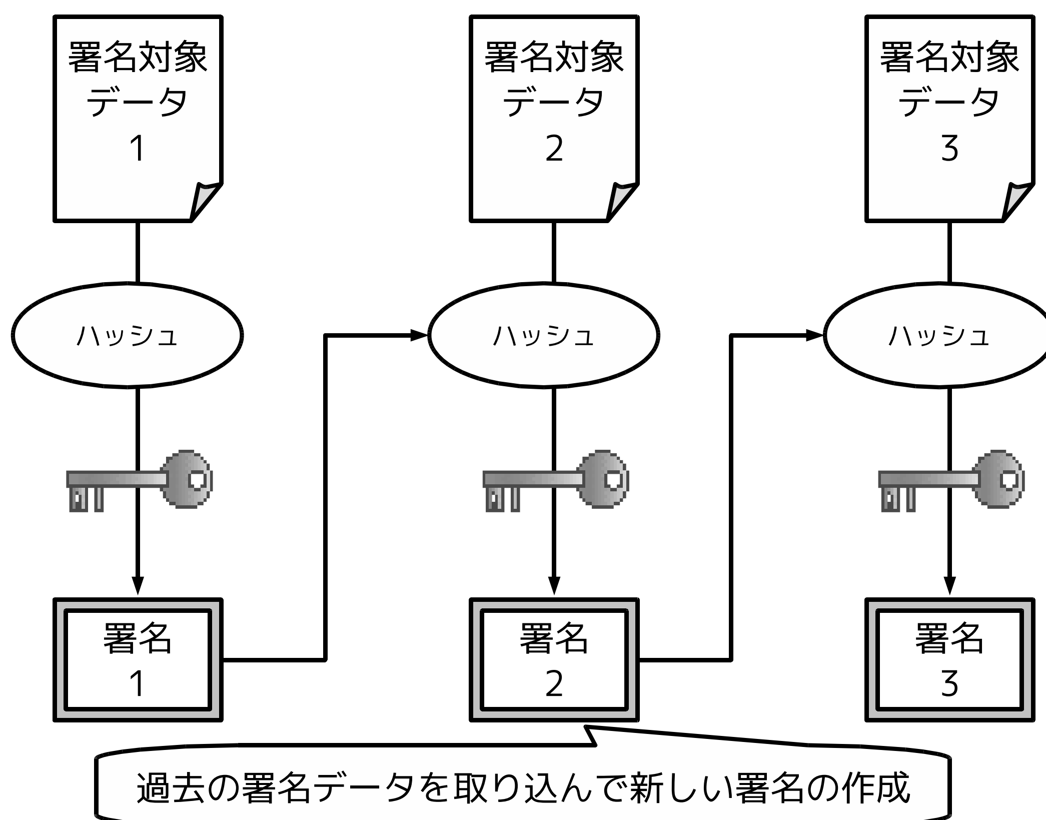

図 2.1: ヒステリシス署名の概要

ヒステリシス署名

ヒステリシス署名とは，デジタル署名の概念を拡張したもので，署名を行う際に，過去に署名したデータを署名対象データに含めた状態で署名処理を行うものである（図 2.1 参照）．ヒステリシス署名は，暗号が危殆化しても署名付きの文書の安全性を確保するために開発されたものである．

文書を登録する度に，過去に生成した署名データを取り込んで，新たな署名を生成するため，各データ間に形成された時系列的な幾重もの連鎖構造をすべて改竄しなければ，文書の改竄をすることができない．そのため，署名の有効性を長期間維持することができ，文書の原本性を容易に確保できる．

表 2.1: 関連研究の特徴

	ログファイル保証の検証	システム管理者の不正防止	ログファイルの改竄防止
WORM への書き込み	×	×	○
逃げログ	△	×	△
署名	△	×	△

2.2.3 関連研究のまとめ

以上で述べた関連研究の特徴のまとめを表 2.1 に示す.

WORM への書き込みによる手法では，ログファイルの改竄を防止することはできるが，ログファイルを保証するための検証やシステム管理者の不正を防止することができない.

逃げログの手法では，ログファイルの改竄は困難である．これは，バックアップを含めたすべてのログファイルを発見されたときにのみログファイルを改竄することができるためである．このことから，ログファイルを保証するための検証も同様にできない可能性があることやシステム管理者の不正を防止することができない.

署名を行う手法では，ログファイルの改竄は困難である．これは，ログファイルの署名していない区間を改竄できるためである．また，ログファイルを 1 つのみ署名できることから一般的に複数存在するログファイルをすべて検証することができないことやシステム管理者の不正を防止できない問題がある.

第3章

提案手法

本章では，ログファイルを保全するために必要な検討事項と提案手法であるログファイルの証拠性を保証する仮想化とヒステリシス署名を用いたログ改竄防止保全システムについて述べる．

3.1 概要

3.1.1 保全のための必要事項

ログファイルは，事件が発生したとき原因を特定するための重要な情報である．そのため，ログファイルを証拠として使用できるように保全しなければならない．ログファイルを保全するには，ログファイルの改竄を防止できること，root 権限の悪用を防止できること，ログファイルの改竄を検出できることが挙げられる．そのため，以下の 3 つのことをすべて満たさなければならないと考えられる．

1. 第三者がログファイルを検証可能

2. root 権限を持つシステム管理者による不正を防止

3. ログファイルの任意の場所の書き換えが不可

ログファイルを裁判の際に証拠として扱うには，第三者がログファイルを検証することで証明する必要がある．そのため，ログファイルが改竄されていないことを証明するシステムが必要になる．

ログファイルは，root 権限を持つシステム管理者（以下，管理者と呼ぶ）が管理する．そのため，管理者は，不正をした痕跡を消去するためにログファイルを書き換えることができる．そこで，root 権限の悪用による不正を防止するためのシステムが必要になる．

一般的にファイルは任意の場所を書き換えることが可能なため，不正をした痕跡を消去するためにログファイルを書き換えることが考えられる．そのため，任意の場所の書き換えを不可能にする必要がある．また，ログファイルは，ログを追記するのみのファイルであるため，ユーザが利用するファイルのように任意の場所を書き換えることを想定する必要はない．

以上のことから，上記 3 つのことをすべて満たすことで，ログファイルを保全できるといえる．

また，一般的にシステムが管理するログファイルは複数存在するため，ログファイルをすべて保全する必要があり，かつ，すべてのログファイルを第三者が証明できるようにする必要がある．

3.1.2 保全システムの実現

3.1.1 で述べた 3 つをすべて満たすことで，ログファイルの信頼性を向上できるため，裁判の際，証拠として扱うことができるようになる．

そこで，ログファイルを保全するために，以下の 3 つを提案する．

1. ヒステリシス署名

2. 仮想化による保全機構

3. 追記処理のみのログファイル

第三者がログファイルを検証できるようにするためには，ヒステリシス署名を用いるという解決策がある．ヒステリシス署名を拡張した署名方式を用いることで，第三者がログファイルを検証することができるため，ログファイルの改竄を検出することが可能となる．システム管理者の不正の問題には，仮想化機構を用いるという解決策

表 3.1: システムの特徴

	ログファイル保証の検証	システム管理者の不正防止	ログファイルの改竄防止
WORM への書き込み	×	×	○
逃げログ	△	×	△
署名	△	×	△
提案システム	○	○	○

がある．仮想化による保全機構を用いることで，システム管理者の不正を防止することができる．ログファイルの任意の場所を書き換えできる問題には，ログファイルを追記のみにするという解決策がある．追記のみにすることによって，ログファイルの任意の場所の書き換えを不可能にする．

関連研究と提案システムのまとめを表 3.1 に示す．関連研究では，ログファイルの満たすべき要点をすべて満たすことができない．特に，システム管理者の不正を防止することができないため，既に満たしている要点を満たさないように改竄することができる可能性がある．提案システムでは，システム管理者の悪用による不正を防止することができるため，他の要点を改竄されることはない．

これで，システムログを記録しているすべてのログファイルを安全に管理し，第三者がログファイルを検証できる新しい保全システムが実現できる．詳細は，3.2 で述べる．追記処理のみのログファイルと仮想化による保全機構で，ログファイルの改竄と管理者の不正を防止し，ヒステリシス署名で，改竄を検出する．保全のための必要事項をすべて満たすことでログファイルを安全に保全することができる．

3.2 全体構成

追記処理のみのログファイルと仮想化を用いたログ保護手法とログファイルの改竄を検出するためのヒステリシス署名を用いたログ保全手法について述べる．提案手法の概要を図 3.1 に示す．

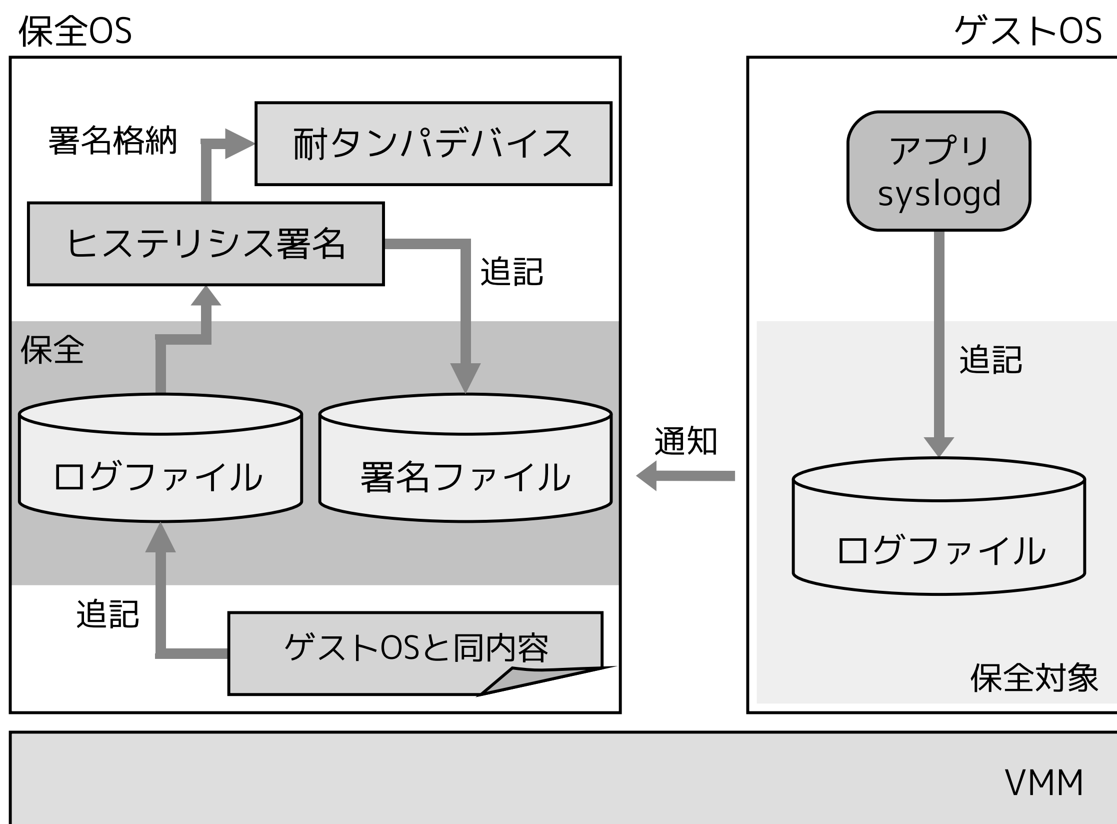

図 3.1: 提案システムの概要

3.2.1　ヒステリシス署名

改竄を検出するためには，ヒステリシス署名を利用する．ヒステリシス署名を用いることで，ログファイルを改竄されていないことを証明することができるようになる．本稿では，ヒステリシス署名の概念を拡張し，複数存在するログファイルに対して同時にヒステリシス署名を行うことと，定期的に署名を行えることをできるようにした．また，最新の署名を耐タンパデバイスへ格納する．そのため，すべてのログファイルの証拠性を保証する．これで，ログファイルに対する改竄を検出することができ，第三者がすべてのログファイルを検証できる．

3.2.2 仮想化による保全機構

仮想化を用いて管理者がログファイルを改竄できなくする．これは，管理者が管理するシステム（以下，ゲスト OS と呼ぶ）と別にログファイルを保管するためのシステム（以下，保全 OS と呼ぶ）を仮想化を用いて用意する．保全 OS とゲスト OS はログを保全するための通信機構を持っており，通信機構を通じてゲスト OS の記録しているログファイルを保全 OS で記録・保持する．また，管理者はゲスト OS のみを利用できる．通信機構を通じてゲスト OS のログファイルを保全 OS においても記録・保持することで，管理者がゲスト OS のログファイルを改竄できなくなり，システム管理者による不正を防止することができる．

3.2.3 追記処理のみのログファイル

ログファイルは，ユーザの利用するファイルとは異なり，任意の場所を書き換える必要がない．そこで，ログファイルに対するアクセス権限を追記のみに制限する．これで，ログファイルの任意の場所を書き換えることが不可能になり，ログファイルの改竄を防止できる．

3.3 ログ保全処理

ヒステリシス署名を利用して，ログファイルを署名し，ログファイルを保全する．保全処理は，保全 OS でのみ行い，署名を耐タンパデバイスに格納する．保全処理を以下で述べる．

3.3.1 複数のログファイルの署名

3.1.1 で述べたが，ログファイルは複数存在するので，すべてのログファイルの証拠性を保証する必要がある．複数のログファイルを署名するために，署名の安全性，検証可能性，実現可能性を検討した．

1 対 1 署名

1 対 1 署名は，ログファイル 1 つに対して署名を 1 つ持つものである．これは，ログファイルを 1 つ 1 つハッシュ値を計算し，ヒステリシス署名を行うため，ログファイルの数に比例して署名に要する時間が大きくなる．そのため，ログファイルの署名を行う間隔が比例して大きくなり，署名の安全性が失われる．

また，ログファイルの数と同じ数だけ署名を管理することになるので，耐タンパデバイスですべての署名を管理できなくなる可能性があるため実現可能性は低い．しかし，ログファイルと署名が 1 対 1 で対応しているため，検証の際，改竄されたログファイルの検出は容易い．以上のことから，1 対 1 署名は，ログファイルを保全するには不適切であると考えられる．

多対 1 署名

多対 1 署名は，複数存在するログファイルすべてに対して署名を唯 1 つ持つものである．これは，すべてのログファイルのハッシュ値に対してヒステリシス署名を行うため，ログファイルの数が増減しても，署名は 1 つのみである．そのため，ログファイルの署名を行う間隔は，ハッシュ値の計算時間のみに比例する．これで，1 対 1 署名よりも署名の安全性は向上し，耐タンパデバイスで管理する署名が 1 つであるため実現可能である．

検証可能性を考慮し，署名ファイルに記録する内容を検討した．検証するためには，ログファイルの原本と署名が必要になる．そこで，署名ファイルには，すべてのログファイルの inode 番号とハッシュ値を記録する．このようにすることで，ログファイルの改竄を検出できる．以上のことから，多対 1 署名を用いることで，ログファイルの証拠性を保証することができる．

3.3.2 高い頻度の署名

ヒステリシス署名は，文書が登録される度に署名を作成するため，定期的な署名の更新を行うことができない．そのため，定期的にログファイルに対してヒステリシス署名を行うことで，ログファイルの信頼性を向上させることができる．

これらのログ保全処理により，署名の検証を行うことで，ログファイルの改竄を検出することができるようになる．また，第三者がログファイルを検証することが可能となり，ログファイルの証拠性が確保され，裁判で証拠として扱うことができるようになる．

3.4 ログ保護処理

ゲストOSでログファイルに対して行われる処理を通信機構を通じて保全OSにおいても処理を行う．行う処理を以下で述べる．

3.4.1 ログ追記処理

一般的に，ログファイルは/var/logディレクトリ以下で管理されている．今後，/var-/logディレクトリ以下のログファイルをログ保全対象ファイルと呼ぶ．ゲストOSがログ保全対象ファイルに対してログを追記したとき，通信機構を通じて保全OSの対応するログファイルに同一のログを追記する．

3.4.2 ログ削除処理

ゲストOSでログ保全対象ファイルの削除を試みようとした場合，ゲストOSでの削除命令を無効化し，削除しようと試みた事実をログとして，ゲストOS，保全OSの両方に追記する．ログ保全対象ファイルは，削除してしまうと証拠性が失われるため，削除は無効化し，削除の事実をログに残す必要がある．

3.4.3 ログ改名処理

ゲストOSでログ保全対象ファイルの名前を変更する場合，通信機構を通じて，保全OS内で管理しているマッピングテーブルから同一のファイル名のものを検索し，マッピングテーブルの対応している名前を変更する．これで，ゲストOSのログファイルと保全OSのログファイルの名前を管理できる．

これらのログ保護処理により，ログファイルを安全に管理することができるようになり，管理者による不正を防止できるようになる．

第4章

実装

提案手法を Linux 上に実装した．通信機構は，仮想マシンモニタ（VMM）である Xen を利用した．ログ保護処理は，カーネルを書き換えることで実現した．また，ヒステリシス署名は，保全 OS 上で行われるように実現した．

4.1 複数のログファイルに対するヒステリシス署名

3.3 で述べた保全処理を実現するために，複数存在するログファイルに対してヒステリシス署名を施すプログラムを作成した．ログファイルを署名する際に，利用した耐タンパデバイスと暗号化 API である PKCS#11 について述べる．その後，複数のログファイルを署名する際に，検討した事項について述べ，署名の流れについて述べる．

4.1.1 耐タンパデバイス

耐タンパデバイスは，アラジン社の eToken PRO[13]（以下，eToken と呼ぶ）を用いた．eToken は，耐タンパ領域を備えており，その内部に秘密鍵と公開鍵を保存することができる USB 型のセキュリティデバイスである．また，秘密鍵を耐タンパ領域から取り出すことができないセキュアデバイスであるが耐タンパ領域は 32K と小さい．また，RSA の公開鍵暗号機能を利用することができる．さらに，標準 Crypto API および，PKCS#11[14] をサポートしているため，アプリケーションから暗号トークンインタフェースを利用することで eToken にアクセスすることができる．

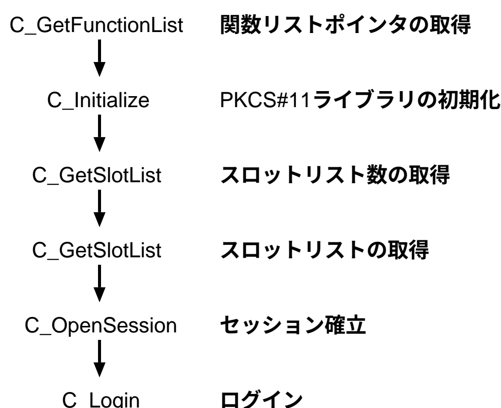

図 4.1: PKCS#11 の初期処理

4.1.2 PKCS#11

PKCS#11とは，暗号トークンへの汎用インタフェースを定義するAPIのことで，eToken などで利用されている耐タンパ領域にアクセスするために必要となる．PKCS#11 を利用して秘密鍵，公開鍵の鍵ペアの作成や署名の作成，検証ができる．本稿では，利用した API と鍵作成や署名作成の流れを述べる．

初期処理

PKCS#11 ライブラリを使用するには，図 4.1 に示すように PKCS#11 ライブラリを初期化する必要がある．まず，C_GetFunctionList 関数で，PKCS#11 ライブラリの関数リストのポインタを取得する．取得した関数リストの中から，ライブラリを使用することが可能となる．C_Initialize 関数は，取得した PKCS#11 ライブラリを初期化する．C_GetSlotList 関数は，1 回目の呼出しでシステムに接続されているトークンの数を取得し，2 回目の呼出しでトークンのリストを取得する．C_OpenSession 関数は，C_GetSlotList 関数で取得したスロット ID に接続されているトークンとセッションを確立する．今後，この関数で取得したセッションハンドルを用いてトークンとアプリケーション間で，PKCS#11 ライブラリを用いた情報をやり取りする．そして，C_Login 関数でトークンにログインする．

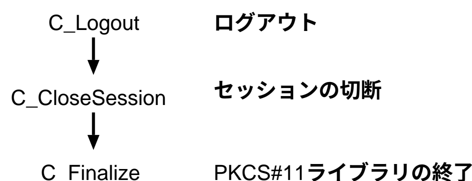

図 4.2: PKCS#11 の終了処理

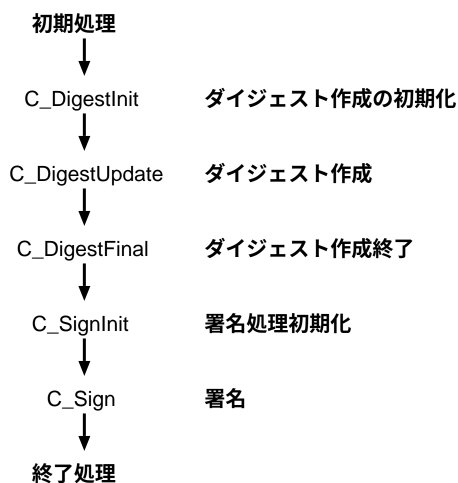

図 4.3: PKCS#11 を用いた署名生成の流れ

終了処理

PKCS#11 ライブラリの使用を終了するには，図 4.2 に示す手順を行う．終了処理は，C_Logout 関数でトークンからログアウトし，C_CloseSession 関数で，トークンとアプリケーション間のセッションを切断する．C_Finalize 関数は，今まで使用してきた PKCS#11 ライブラリを終了する．これらの初期処理と終了処理は，トークンを利用する際に，行う必要がある。

鍵ペア作成

PKCS#11 ライブラリを使用して，公開鍵と秘密鍵の鍵ペアを作成するには，C_GenerateKeyPair 関数を用いる．この関数は，鍵作成のメカニズムを指定することで，様々な種類の鍵を作成することができるものである．本システムでは，安全性を考慮し，公開鍵暗号方式の一つである RSA を用いた．

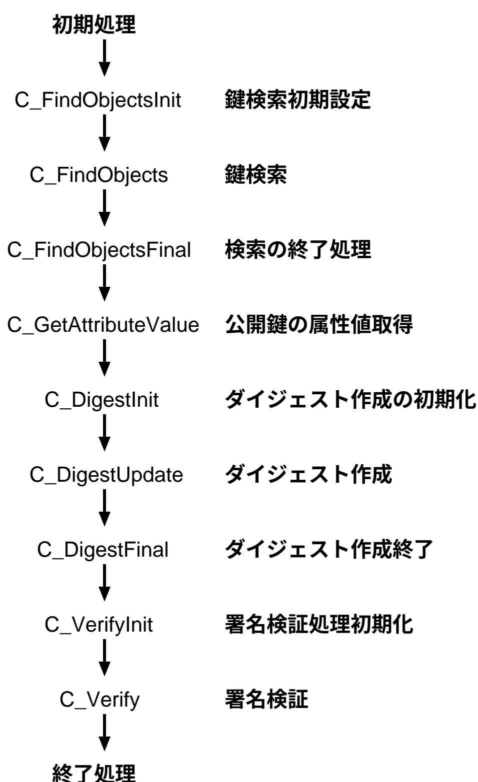

図 4.4: PKCS#11 を用いた署名検証の流れ

署名生成

署名生成の流れを図 4.3 に示す．C_DigestInit 関数は，ハッシュ値を計算するためのメカニズムを設定する．本システムのハッシュ値を計算するメカニズムは，SHA-1 とする．C_DigestUpdate 関数は，設定したメカニズムを用いて，署名の対象となるデータのハッシュ値を計算する．C_DigestFinal 関数は，C_DigestUpdate 関数で計算したハッシュ値を格納する．C_SignInit 関数は，署名処理の初期化を行う．初期化は，署名のメカニズムを設定する．本システムは，RSA 方式の鍵を用いるため，署名のメカニズムを RSA に設定する．C_Sign 関数は，設定した署名メカニズムを用いて，C_DigestFinal 関数で作成したハッシュ値を署名する．この署名生成に用いた PKCS#11 ライブラリの関数を用いて，4.1.5 で述べる複数のログファイルに対するヒステリシス署名を行う．

署名検証

生成した署名を検証するための流れを図 4.4 に示す。鍵ペア作成の処理で作成した公

開鍵，秘密鍵の鍵ペアから公開鍵を取得する．まず，鍵の属性値を C_FindObjectsInit 関数で設定する．本システムでは，鍵にラベルを付加しているので，属性値のラベルを設定する．C_FindObjects 関数で，設定した属性値のオブジェクトを検索する．本システムでは，公開鍵，秘密鍵に設定した属性値が付加されているので，これらのオブジェクトが検索される．C_FindObjectsFinal 関数で，オブジェクトの検索処理を終了する．

C_DigestInit 関数から C_DigestFinal 関数までの処理は署名生成と同様に行う．C_VerifyInit 関数は，作成した署名を検証するためのメカニズムを設定する．本システムは，署名生成と同様に RSA を用いる．C_Vefiry 関数は，署名を公開鍵で復号した値と計算したダイジェストの値を比較する．署名検証で用いた PKCS#11 ライブラリの関数を用いて，4.2 で述べるヒステリシス署名で作成した署名を検証する．

これらの PKCS#11 ライブラリを使用した処理を用いて 4.1.5 や 4.2 で述べる署名作成や検証を行う．

4.1.3　ログファイルの自動分割

多対 1 署名で，署名の間隔を短縮することができるようになるが，ログファイルのハッシュ値を計算する時間を短縮することも検討する必要がある．ログファイルのハッシュ値を計算する時間は，ファイルサイズに比例して大きくなるため，ファイルサイズを小さくする必要がある．そこで，ファイルサイズを一定サイズで自動的に分割する方法を提案する．分割サイズは，ハッシュ値計算の時間を可能な限り短くすることとファイル管理のバランスを考慮して 10MB に設定した．分割方法を図 4.5 に示し，ファイルの管理手順を述べる．

4.1.4　署名列

ヒステリシス署名を行った結果を署名ファイルに記録する署名列の構成を述べる．署名列は，従来のヒステリシス署名の署名データを拡張したものである．署名列の構造を表 4.1 に示す．

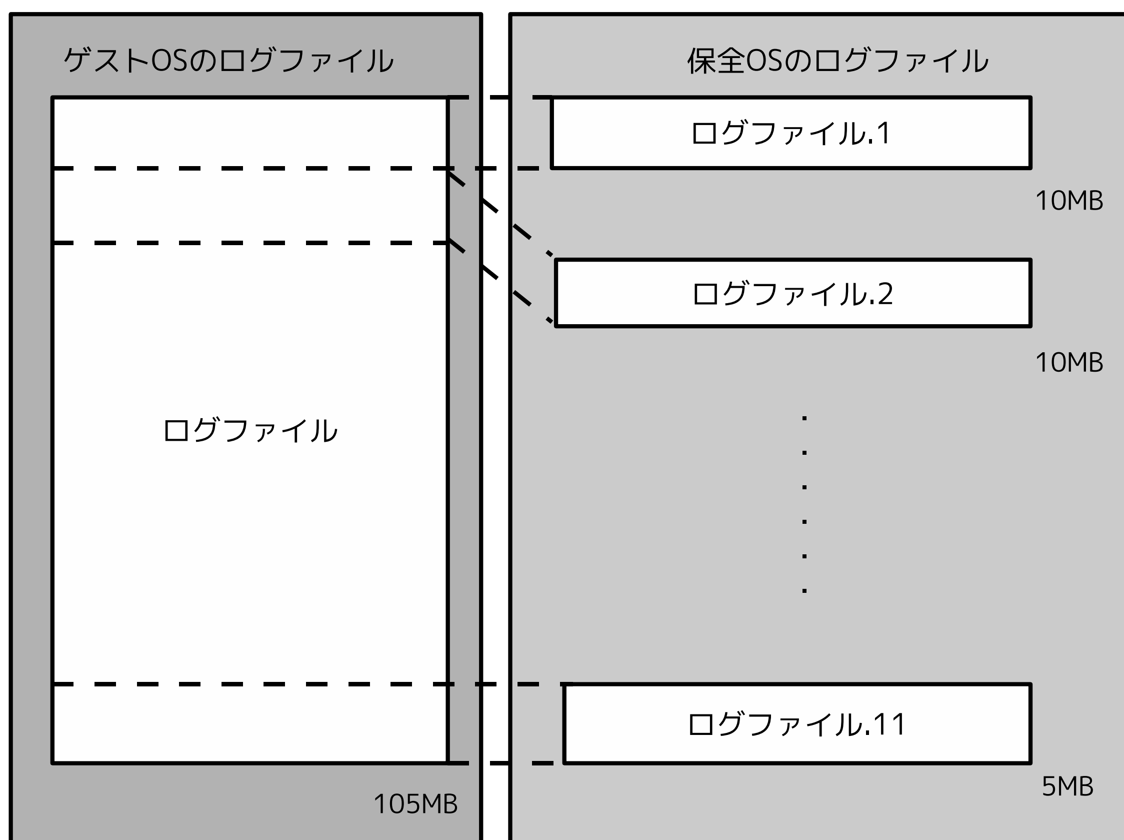

図 4.5: ログファイルの自動分割方法

表 4.1: 署名列の構造

ファイル数	inode 番号	連鎖	ハッシュ値	署名

ファイル数は，ヒステリシス署名の対象となるログファイルの数のことで，inode 番号は，ファイルの固有識別子のことである．連鎖は，1 回前のヒステリシス署名した結果の署名列のハッシュ値のことである．ハッシュ値は，ログファイルのハッシュ値のことで，署名は，ハッシュ値を秘密鍵で暗号化したものである．inode 番号とハッシュ値は，ファイル数に比例して増減する．このような構造の署名列を記録することは，?? で述べた検証可能性を考慮したためである．これで，いつログファイルが改竄されたかやどのログファイルを改竄されたかを検出することができるようになる．

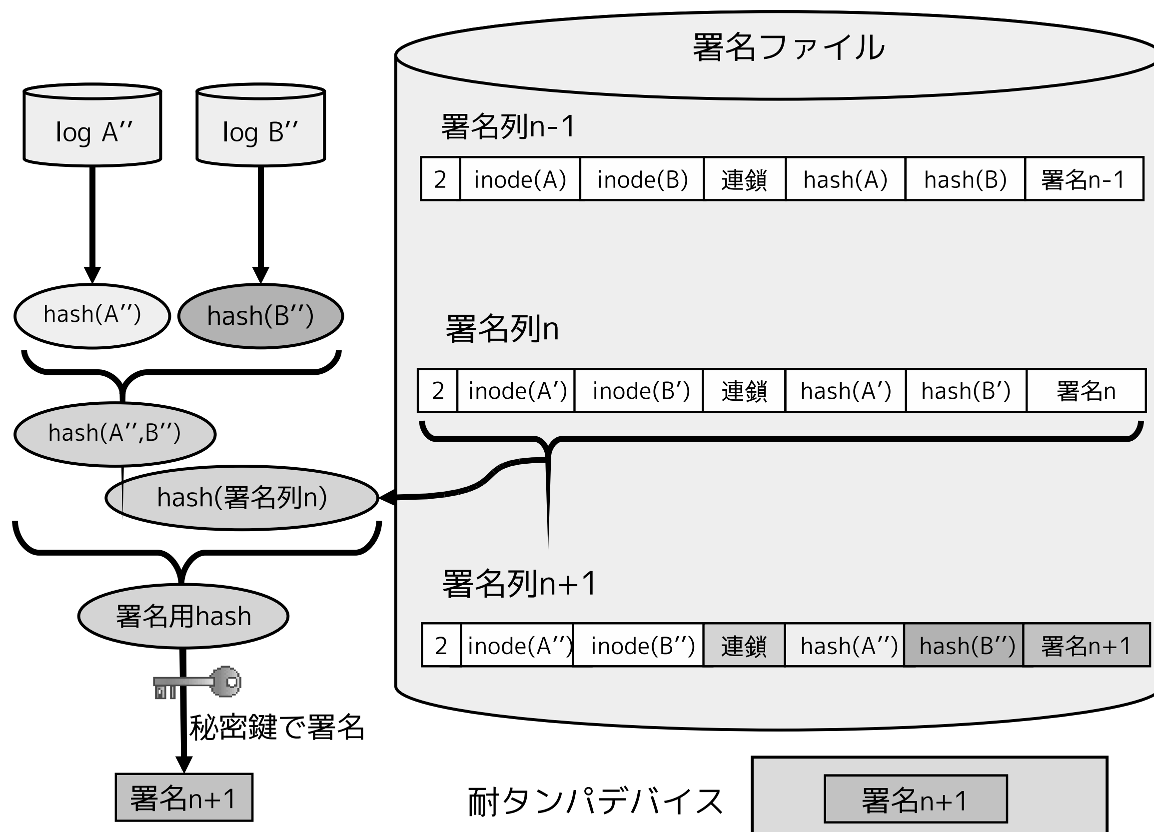

図 4.6: 複数ログファイルのヒステリシス署名の処理

4.1.5 署名手順

複数のログファイルに対してヒステリシス署名を行う処理を例として図 4.6 に示し，手順を述べる．

最初に，ヒステリシス署名の対象となるログファイルをすべて取得する．ここでは，ログファイル log A”，log B” の 2 つを対象とする．また，署名した結果を記録する署名ファイルを取得する．署名ファイルには，既に 1 回前と 2 回前に署名した結果である署名列 n-1，署名列 n が記録されているものとする．それでは，署名列 n+1 を求める手順について述べる．

log A”，log B” をそれぞれハッシュ値を計算する．計算したハッシュ値をそれぞれ hash(A”)，hash(B”) とする．hash(A”)，hash(B”) を連結し，ハッシュ値 hash(A”, B”)

を計算する．また，現在の最新の署名結果である署名列nからハッシュ値hash(署名列n)を計算する．ここでは，便宜上，hash(署名列n)は，署名列nから計算するとあるが，実際には，署名列n中の署名nは，耐タンパデバイスに格納されている署名nを利用する．その後，hash(A”, B”)とhash(署名列n)から署名用ハッシュ値を計算し，署名用hashをeTokenに格納されている秘密鍵を用いて暗号化を行い署名n+1を求める．

署名した結果を署名列n+1として記録する．署名列n+1には，ログファイルの数である2，それぞれのファイルの固有識別子であるinode(A”)，inode(B”)，連鎖のhash(署名列n)，ログファイルのハッシュ値であるhash(A”)，hash(B”)，署名n+1を記録する．

しかし，署名ファイルに署名列を記録するだけでは，署名ファイルを改竄されても検出することができない．署名ファイルの改竄を防止するために最新の署名をeTokenにも保管する必要がある．これにより，署名ファイルの改竄を防止できる．検証の手順は，4.2で述べる．

4.2　署名検証

4.1.5で述べたヒステリシス署名を行い，記録した署名を検証する手順を図4.7に示し，処理の流れを述べる．ここでは，2つのログファイルの正当性を検証する．

最初に，検証の対象となるログファイルと署名ファイルを取得する．また，署名ファイルから最新の署名列nを取得する．検証するログファイルlog A，log Bからファイルの固有識別子であるinode番号を取得する．署名列nに記録されているinode(A)，inode(B)と比較し，ログファイルが検証の対象となっているファイルか確認する．また，署名列のファイル数と取得したログファイルの実際の数が異なる場合，ログファイルが削除されたことを検出する．

ログファイルlog A，log Bが改竄されていないことを検証する．この検証では，log A，log Bからハッシュ値hash(A)，hash(B)を計算し，署名列nに記録されているhash(A)，hash(B)と比較する．一致している場合，ログファイルは改竄されていないことを確認できる．一致しない場合，log A，log Bのどちらかが改竄されていることを検証者に示し，検証を終了する．

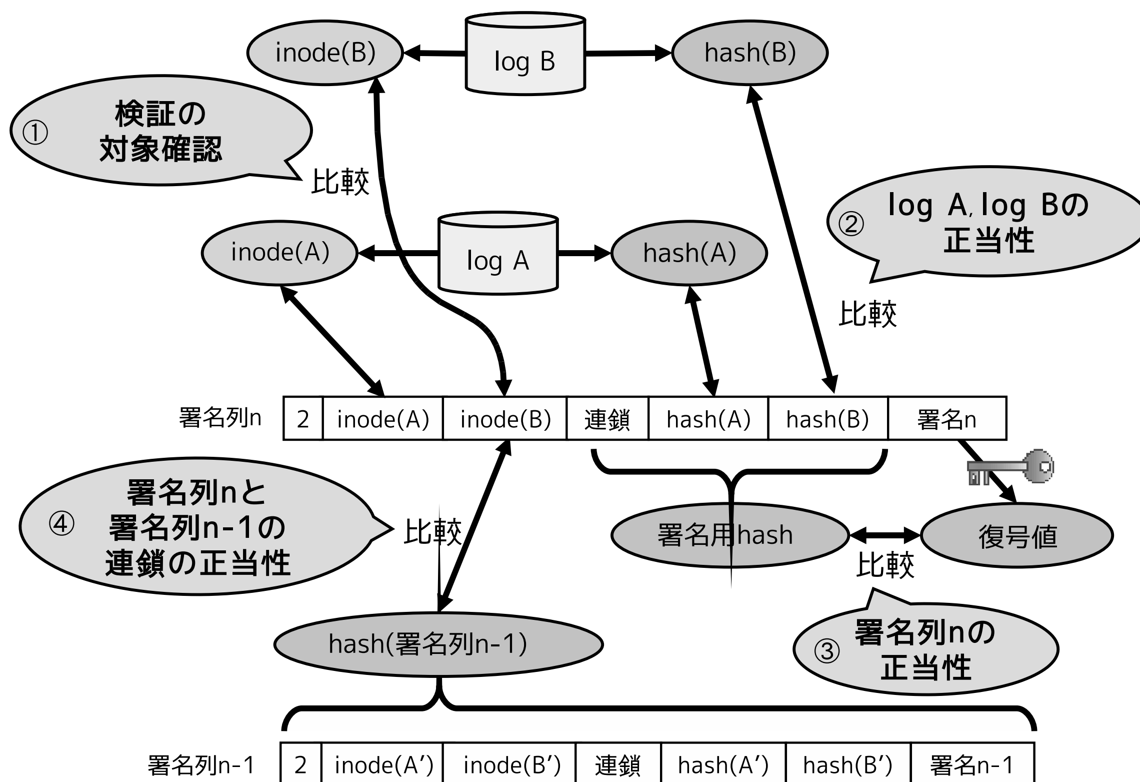

図 4.7: 署名検証の流れ

署名列 n が改竄されていないことを検証する．この検証では，eToken に格納されている署名と署名列 n に記録されている署名 n を比較する．一致する場合，署名列 n が削除されていないことを確認できる．また，署名列に記録されている連鎖，hash(A), hash(B) から署名用のハッシュ値を計算し，署名 n を eToken に格納されている公開鍵で復号した値と比較する．一致する場合，署名列 n は，改竄されていないことを確認できる．一致しない場合，署名列が改竄されていることを検証者に示し，検証を終了する．

最後に，署名列間で連鎖して署名されていることを検証する．この検証では，過去から現在に至るまでに連鎖して署名されているか検証する．署名列 n-1 からハッシュ値 hash(署名列 n-1) を計算し，署名列 n に記録されている連鎖の値を比較する．一致している場合，署名列 n と署名列 n-1 の間で連鎖して署名されていることを確認でき

る．一致しない場合，署名列 n と署名列 n-1 は連鎖して署名されていないことを検証者に示し，検証を終了する．その後，署名列の検証と連鎖を遡って署名ファイルの先頭まで検証する．

以上の手順で，ログファイルと署名ファイルに対する検証を行う．すべての手順で検証が正しい場合は，ログファイルと署名ファイルの正当性を確認することができる．しかし，いずれかの場合で検証が正しくない場合は，ログファイルもしくは，署名ファイルが改竄されていることを意味し，正当性を確認することができない．

4.3 保全通信機構を用いた保護処理

仮想化を用いて保全 OS とゲスト OS が通信するための機構を実装し，通信機構を通じて保全 OS が 3.4 で述べた保護処理を行う．これは，ゲスト OS の呼び出すシステムコールを書き換えることで保護処理を実現した．まず，実現する際に用いた VMM である Xen について述べ，書き換えたシステムコールについて述べる．

4.3.1 Xen

Xen とは，コンピュータを仮想化し，複数の OS を同時並行に動作させられるようにする仮想化ソフトの 1 つである．2002 年にイギリスのケンブリッジ大学コンピュータ研究室の Ian Pratt 氏らが開発を始めたもので，現在ではオープンソースソフトとして公開されており，XenSource 社 [15] が管理している．

Xen は，仮想マシンモニタ（Virtual Machine Monitor:VMM）と呼ばれるソフトウェアの 1 つである．コンピュータのハードウェア資源を一括で管理し，OS は仮想的なコンピュータとして振舞うものである．

本稿では，準仮想化と呼ばれる動作モードでログ保全の通信機構を実現した．通信機構は，OS 間で共有される空間を介して行われる．共有する空間は，設定情報などを格納する XenStore や共有メモリが存在する．本システムでは，共有メモリを介して情報をやりとりする方法を実装した．

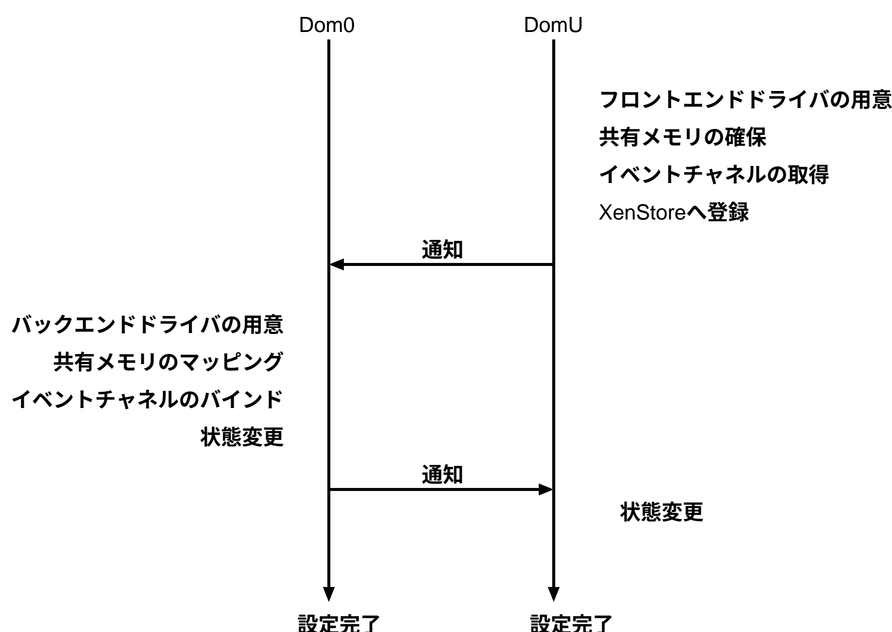

図 4.8: Xen の設定アルゴリズム

4.3.2 通信機構

ログを保全するための通信機構は，Xen 独自の VMM を利用して実現した．通信機構を利用するための設定アルゴリズムを図 4.8 に示す．図 4.8 に示す Dom0，DomU はそれぞれ本システムの保全 OS，ゲスト OS に対応する．

まず，ゲスト OS が通信用のドライバ (以後，ゲスト OS 側の通信用のドライバをフロントエンドドライバと呼ぶ) を用意する．ゲスト OS が保全 OS と情報をやりとりするために OS 間の共有メモリを確保し，通信用のイベントチャネルを取得し，ゲスト OS のマッピングテーブルを XenStore に登録する．保全 OS は，XenStore に登録された内容からゲスト OS からの共有メモリの参照番号とイベントチャネルを取得する．そして，通信用のドライバ（以後，保全 OS 側の通信用のドライバをバックエンドドライバと呼ぶ）を用意し，これらの情報を用いて，ゲスト OS のメモリのマッピングやイベントチャネルのバインドを行う．これらの処理が行われたらゲスト OS と通信が可能な状態へと自身の状態を変更し，現在の通信状態を保持する Xenbus に登録されている自身の状態を通信可能な状態へ変更する．ゲスト OS は，保全 OS が通信可能な状態へと変更されたときに自身の状態も同様に通信可能な状態へ変更し同様に Xenbus

```
struct sh_info{ // 共有メモリ
  unsigned int    flag;    // 関数フラグ
  unsigned int    res;     // Dom 0 待ちフラグ
  unsigned int    channel; // イベントチャネル
  struct ftdev_op frop;    // 共有メモリのデータ構造
}

struct ftdev_op{ // ファイル操作構造体
  int          op;         // ファイル操作の ID
  int          fd;         // ファイルディスクリプタ
  unsigned int flags;      // open 時のファイルのフラグ
  int          mode;       // open 時のファイルのアクセスモード
  char         path[256];  // ファイル名
  char         buff[2048]; // バッファ
}
```

図 4.9: 共有メモリの構造

に登録されている状態を変更する．Xenbus に登録されている両 OS の状態が通信可能な状態となることで，ゲスト OS と保全 OS 間で通信が可能となる．

本システムで用いる OS 間の共有メモリの構造を図 4.9 に示す．sh_info 構造体は，通信用のドライバの情報を保持するものである．flag は，ゲスト OS で呼ばれたシステムコールの ID を保持するものである．res は，op は，どの種類のファイル操作がゲスト OS で行われたかを保持するもので，path は，操作の対象となったファイル名を保持する．また，buff は，書き込んだログのを保持するものである．4.3.8 で述べるが buff は，変更後のファイル名を保持することになる．今後，各システムコールが通信用のデバイスに渡すデータをファイル操作構造体と呼ぶ．

```
while(バックエンドドライバから読み取り)
  switch(ファイル操作の ID){
    case 0 : open;   // ログファイルを開く
    case 1 : write;  // ログの追記
    case 2 : unlink; // 削除しようとしたログを記録
    case 3 : rename; // マッピングテーブルの更新
    case 4 : close;  // ログファイルを閉じる
  }

  write(バックエンドドライバに書き込み)
}
```

図 4.10: 保全デーモンの処理アルゴリズム

4.3.3 保全 OS の保全デーモン

保全 OS 内に存在する保全デーモンは，バックエンドドライバから，ソフトウェア割り込みの通知を受け取ったときに，3.4 で述べた保護処理を行うものである．この処理を行うアルゴリズムを図 4.10 に示す．保全デーモンは，バックエンドドライバから割り込み通知を受け取るとデバイスファイルからデータを読み取り，ファイル操作構造体に格納する．ファイル操作構造体のファイル操作の ID 別にそれぞれの関数の処理を行う．それぞれの関数の処理を終了するとバックエンドドライバに書き込みを行い，フロントエンドドライバに処理が終了したことを通信機構を通じて通知することで処理が完了する．それぞれの関数の処理は 4.3.4 以降で述べる．また，保全デーモンは，ゲスト OS 側で管理するログファイルのファイル名と保全 OS 側で管理するログファイルのファイル名た対応するマッピングテーブルを保持する．

4.3.4 ログファイルを開く処理

ログファイルを開く処理は，ゲスト OS でログ保全対象ファイルを開いた場合に，通信機構を通じて保全 OS 内の保全デーモンに通知し，ランダムなファイル名のものを開

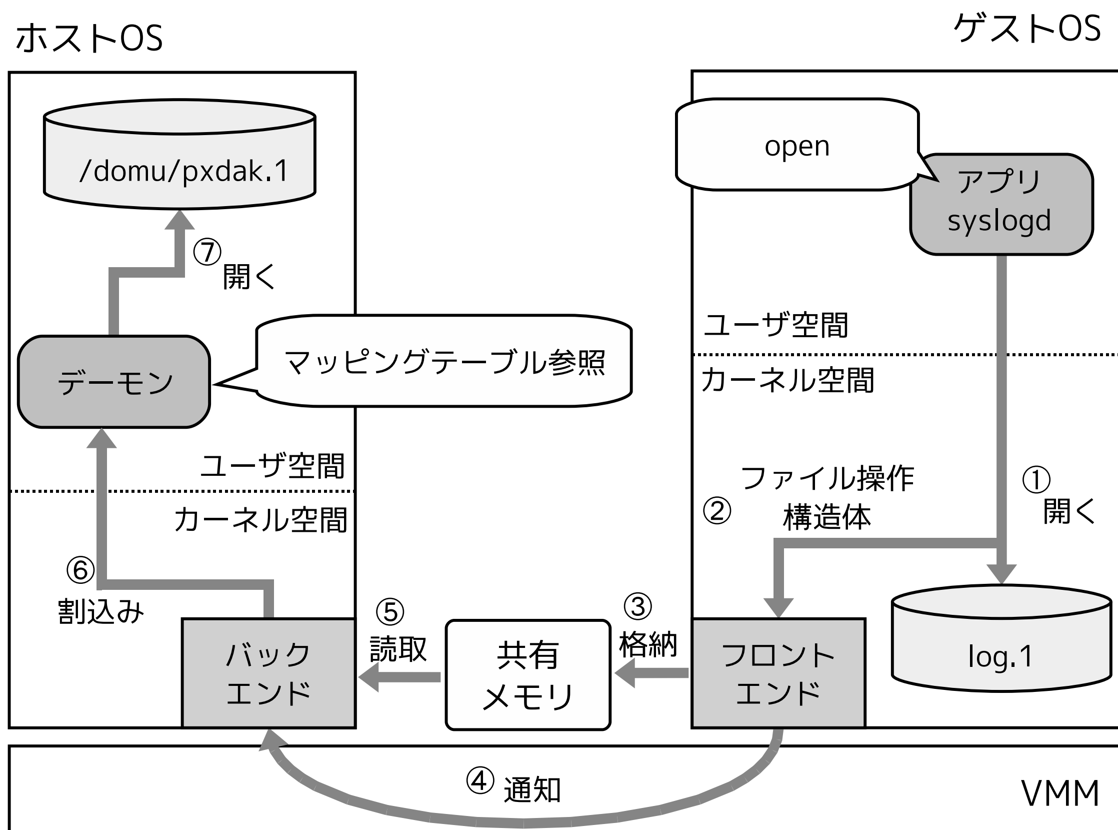

図 4.11: open システムコールの動作

くものである．ログファイルを開く処理の流れを図 4.11 に示し，動作の流れを述べる．

(1) アプリケーションや syslog デーモンが保全対象ファイルに対して open システムコールを呼び出す．ゲスト OS 側の open システムコールの処理を図 4.12 に示す．open システムコールは，引数に開くファイルのパス名，ファイルのアクセスモード，ファイル状態フラグを指定する．ここでは，ログ保全対象ファイルの log.1 を開くとする．このとき，ファイルのアクセスモードに追記のアクセス権限を追加する．

(2) ファイル操作構造体を図 4.12 で示すように各種設定し，フロントエンドドライバへファイル操作構造体を渡す．

(3) フロントエンドドライバは，渡されたファイル操作構造体を保全 OS とゲスト OS

```
sys_open(const char __user *filename, int flags, int mode){
  if(ログ保全対象ファイルである){
    flags = flags | O_APPEND; // 書き込みを追記に設定

    fd = open(filename, flags, mode); // ゲストOS側のopen処理

    // ファイル操作構造体の設定
    ftdev_op->op    = 0;  // ファイル操作のID
    ftdev_op->fd    = fd; // 取得したファイルディスクリプタ
    ftdev_op->flags = flags;
    ftdev_op->mode  = mode;
    ftdev_op->path  = filename;

    フロントエンドドライバにファイル操作構造体を渡す
  }
}
```

図 4.12: ゲスト OS の open システムコールの処理アルゴリズム

間の共有メモリに書き込む．

(4) ゲスト OS は VMM を通じて保全 OS にイベントを通知する．

(5) ゲスト OS からのイベント通知を受け取ったバックエンドドライバは，共有メモリからファイル操作構造体を読み込む．

(6) 保全 OS 内のユーザ空間に存在する保全デーモンにソフトウェア割り込みを通知する．

(7) 保全デーモンは，バックエンドドライバからファイル操作構造体を読み取り，ランダムなファイル名 pxdak.1 を開き，マッピングテーブルに log.1 と pxdak.1 の対応を記録する．

このような流れで，ログファイルを開く処理が行われる．

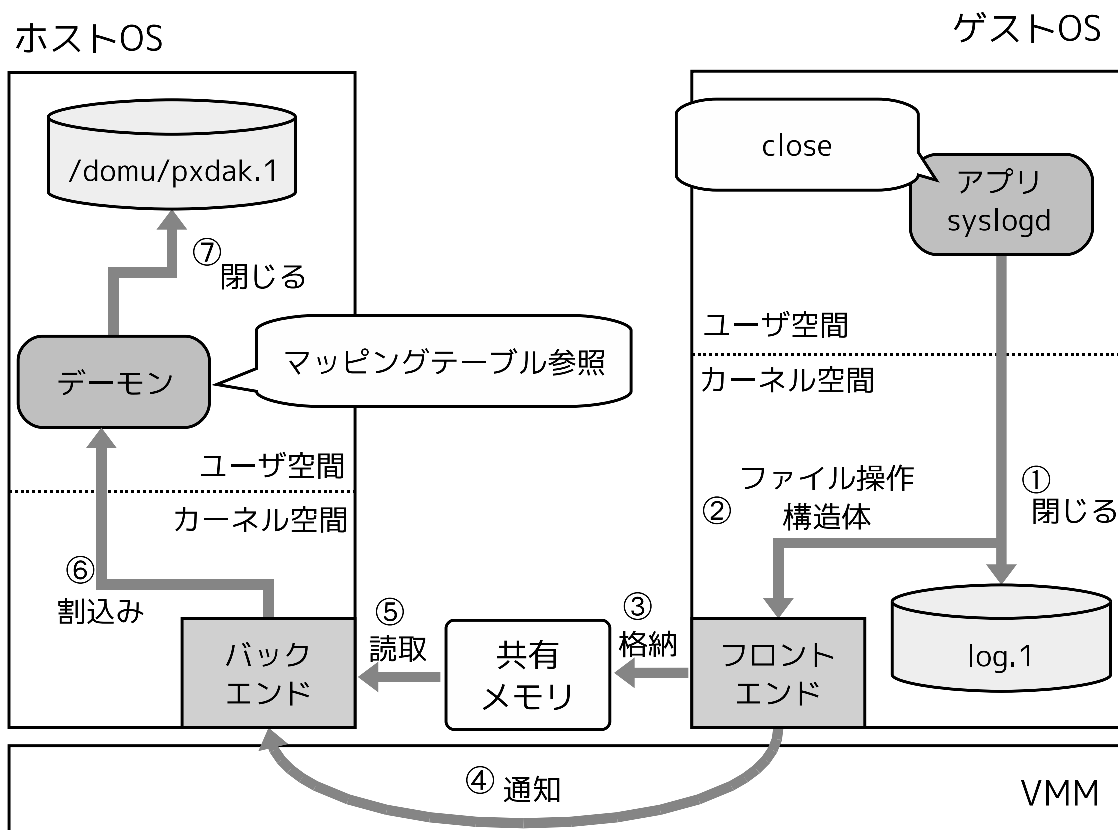

図 4.13: close システムコールの動作

4.3.5　ログファイルを閉じる処理

ログファイルを閉じる処理は，ゲスト OS でログ保全対象ファイルを閉じた場合に，通信機構を通じて保全 OS 内の保全デーモンに通知し，対応するファイルを閉じるものである．ログファイルを閉じる処理の流れを図 4.13 に示し，動作の流れを述べる．

(1) アプリケーションや syslog デーモンが保全対象ファイルに対して close システムコールを呼出す．ゲスト OS 側の close システムコールの処理を図 4.14 に示す．close システムコールは，引数に open システムコールで取得したファイルディスクリプタを指定する．ここでは，ログファイル log.1 のファイルディスクリプタの値である．

(2) ファイル操作構造体を図 4.14 に示すように各種設定し，フロントエンドドライバ

```
sys_close(int fd){
  close(fd);

  // ファイル操作構造体の設定
  ftdev_op->op = 4;  // ファイル操作の ID
  ftdev_op->fd = fd; // 閉じるファイルディスクリプタ

  フロントエンドドライバへファイル操作構造体を渡す
}
```

図 4.14: ゲスト OS の close システムコールの処理アルゴリズム

へファイル操作構造体を渡す.

(3) フロントエンドドライバは，渡されたファイル操作構造体を保全 OS とゲスト OS 間の共有メモリに書き込む.

(4) ゲスト OS は VMM を通じて保全 OS にイベントを通知する.

(5) ゲスト OS からのイベント通知を受け取ったバックエンドドライバは，共有メモリからファイル操作構造体を読み込む.

(6) 保全 OS 内のユーザ空間に存在する保全デーモンにソフトウェア割り込みを通知する.

(7) 保全デーモンは，バックエンドドライバからデータを読み取り，対応するファイル pxdak.1 を閉じる.

このような流れで，ログファイルを閉じる処理が行われる.

4.3.6 ログ書き込み処理

ログ書き込み処理は，ゲスト OS でログ保全対象ファイルに追記を行った場合に，通信機構を通じて保全 OS が対応するログファイルに対して追記を行うものである．ロ

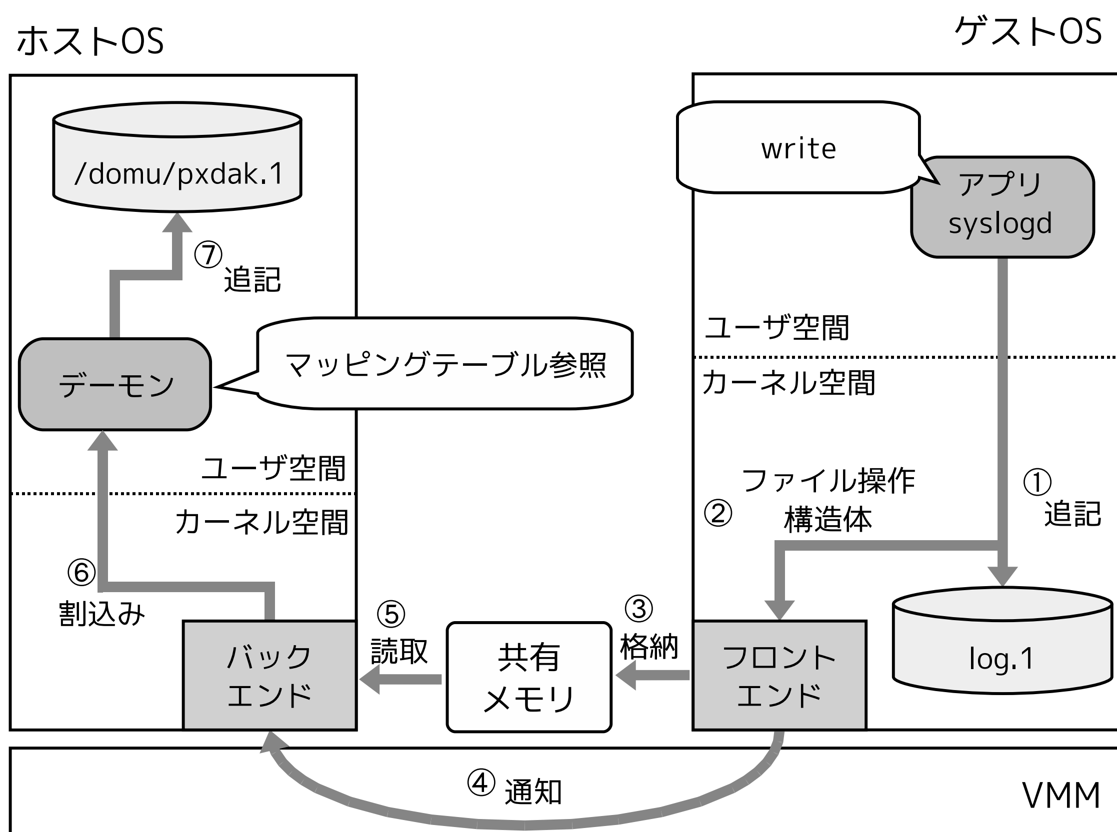

図 4.15: write システムコールの動作

グ書き込み処理の流れを図 4.15 に示し，動作の流れを述べる．

(1) アプリケーションや syslog デーモンが保全対象ファイルに対して write システムコールを呼出す．ゲスト OS 側の write システムコールの処理を図 4.16 に示す．write システムコールは，open システムコールで取得したファイルディスクリプタと書き込む内容のバッファと書き込む内容のサイズを引数とするものである．ここでは，ログファイル log.1 にログを書き込むとする．

(2) ファイル操作構造体を図 4.16 に示すように各種設定し，フロントエンドドライバへファイル操作構造体を渡す．

(3) フロントエンドドライバは，渡されたファイル操作構造体を保全 OS とゲスト OS 間の共有メモリに書き込む．

```
sys_write(int fd, const char __user *buf, size_t count){
  write(fd, buf, count); // ゲスト OS のファイルに buf を書き込む

  // ファイル操作構造体の設定
  ftdev_op->op   = 1;   // ファイル操作の ID
  ftdev_op->fd   = fd;  // 書き込み先のファイルディスクリプタ
  ftdev_op->buff = buf;

  フロントエンドドライバへファイル操作構造体を渡す
}
```

図 4.16: ゲスト OS の write システムコールの処理アルゴリズム

(4) ゲスト OS は VMM を通じて保全 OS にイベントを通知する．

(5) ゲスト OS からのイベント通知を受け取ったバックエンドドライバは，共有メモリからファイル操作構造体を読み込む.

(6) 保全 OS 内のユーザ空間に存在する保全デーモンにソフトウェア割り込みを通知する．

(7) 保全デーモンは，バックエンドドライバからファイル操作構造体を読み取り，対応するファイル pxdak.1 にファイル操作構造体の buff を書き込む.

このような流れで，ログの書き込み処理が行われる．

4.3.7 ログファイルの削除処理

3.4 で述べた削除処理を実現するために，ゲスト OS の unlink システムコールを書き換えた．ログファイルの削除処理は，ゲスト OS でログ保全対象ファイルの削除を無効化し，ログ保全対象ファイルを削除しようとした事実をログとして，ゲスト OS，保全 OS に同一の内容を追記するものである．ログファイルの削除処理の流れを図 4.17 に示し，動作の流れを述べる．

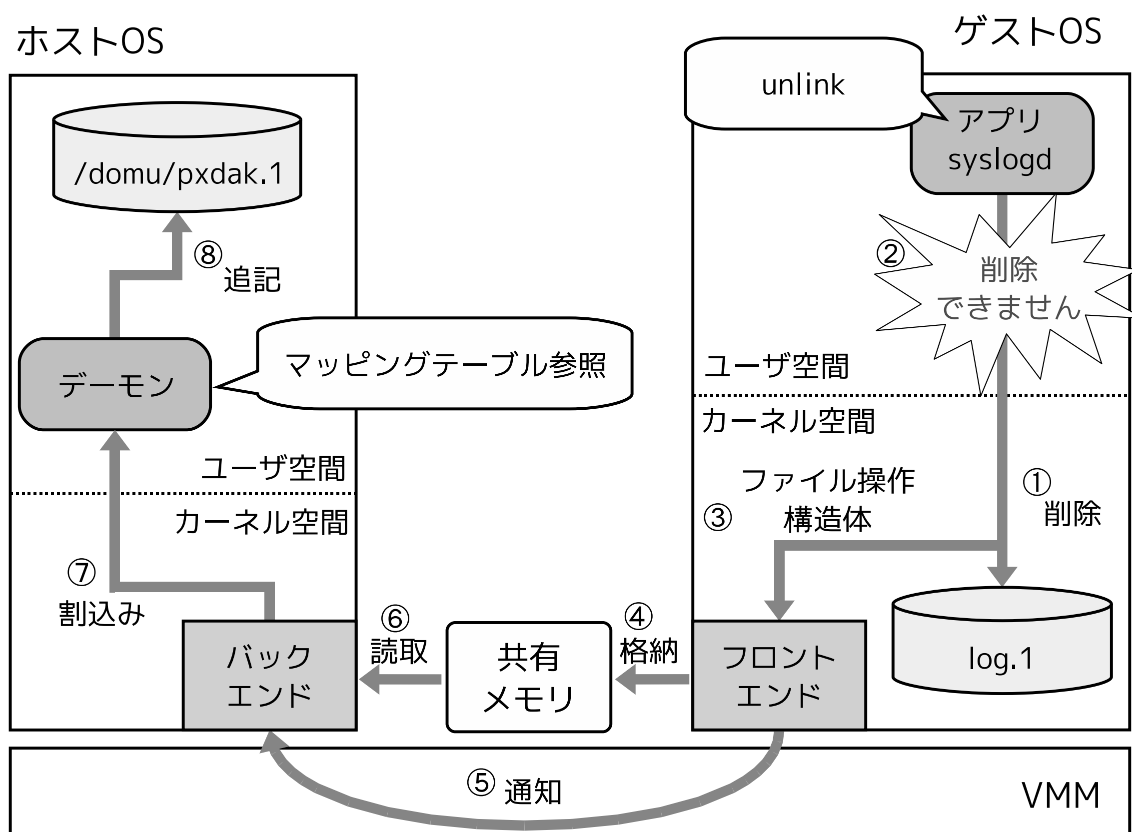

図 4.17: unlink システムコールの動作

(1) アプリケーションや syslog デーモンが保全対象ファイルに対して unlink システムコールを呼出す．ゲスト OS 側の unlink システムコールの処理を図 4.18 に示す．unlink システムコールは，削除するファイルのパス名を引数とするものである．ここでは，ログファイル log.1 を削除しようとする．

(2) ログ保全対象ファイルは削除が無効化されるため，ユーザには,「このファイルを削除することはできません」と通知する．

(3) ログファイル log.1 に削除しようとしたファイル名と事実をログファイルに追記し，ファイル操作構造体に図 4.18 で示すように各種設定し，フロントエンドドライバにファイル操作構造体を渡す．

(4) フロントエンドドライバは，渡されたデータをファイル操作構造体に書き込む．

```
sys_unlink(const char __user *filename){
  if(filename == ログ保全対象ファイル) {
    ユーザに「このファイルは削除できません」の通知

    // filename に削除しようとした事実を記録
    write(filename, 「filename を削除しようとしました」)

    // ファイル操作構造体の設定
    ftdev_op->op   = 2; // ファイル操作の ID
    ftdev_op->buff = 「filename を削除しようとしました」;

    フロントエンドドライバへファイル操作構造体を渡す
  }
}
```

図 4.18: ゲスト OS の unlink システムコールの処理アルゴリズム

(5) ゲスト OS は VMM を通じて保全 OS にイベントを通知する.

(6) ゲスト OS からのイベント通知を受け取ったバックエンドドライバは，共有メモリからファイル操作構造体を読み込む.

(7) 保全デーモンにソフトウェア割り込みを通知する.

(8) 保全デーモンは，バックエンドドライバからファイル操作構造体を読み取り，対応するログファイル pxdak.1 にファイル操作構造体の buff を書き込む.

このような流れでログファイルの削除処理が行われる.

4.3.8 ログファイルの名前変更処理

3.4 で述べた改名処理を実現するために，ゲスト OS の rename システムコールを書き換えた．ログファイルの名前変更処理は，ゲスト OS がログ保全対象ファイルの名前

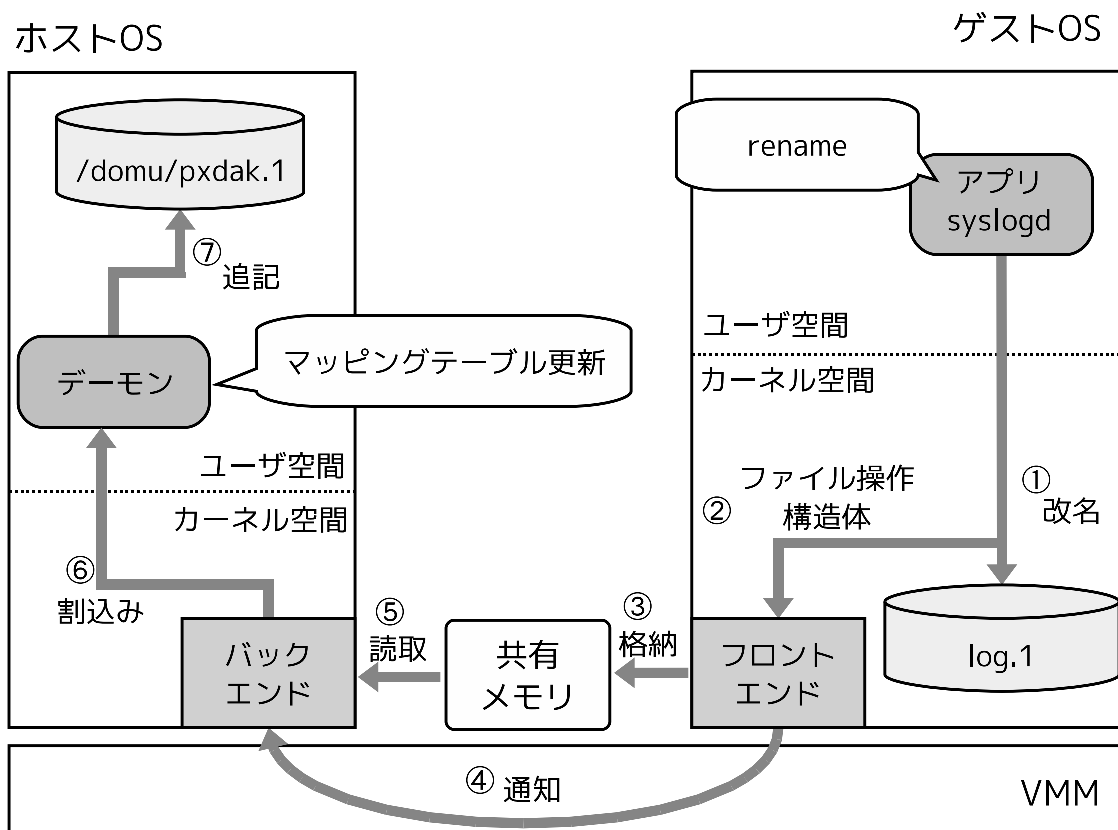

図 4.19: rename システムコールの動作

を変更した場合，ゲスト OS の変更前のファイル名を保全 OS で検索し，対応するファイルをゲスト OS で変更した名前に変更するものである．ログファイルの名前変更処理の流れを図 4.19 に示し，動作の流れを述べる．

(1) アプリケーションや syslog デーモンが保全対象ファイルに対して，rename システムコールを呼出す．ゲスト OS 側の rename システムコールの処理を図 4.20 に示す．rename システムコールは，変更するファイル名と変更後のファイル名ファイル名を引数とするものである．ここでは，ログファイル log.1 からログファイル text.1 に変更するとする．

(2) ファイル操作構造体に図 4.20 で示すように各種設定し，フロントエンドドライバにファイル操作構造体を渡す．

```
sys_rename(const char __user *oldname, const char __user *newname){
  // oldnameからnewnameへ名前の変更
  rename(oldname, newname);

  // ファイル操作構造体の設定
  ftdev_op->op   = 3;       // ファイル操作のID
  ftdev_op->path = oldname; // 変更前のファイル名
  ftdev_op->buff = newname; // 変更後のファイル名

  フロントエンドドライバへファイル操作構造体を渡す
}
```

図 4.20: ゲスト OS の rename システムコールの処理アルゴリズム

(3) フロントエンドドライバは，渡されたファイル操作構造体を共有メモリに書き込む．

(4) ゲスト OS は VMM を通じて保全 OS にイベントを通知する．

(5) ゲスト OS からのイベント通知を受け取ったバックエンドドライバは，共有メモリからファイル操作構造体を読み込む．

(6) 保全デーモンにソフトウェア割り込みを通知する．

(7) 保全デーモンは，バックエンドドライバからファイル操作構造体を読み取り，マッピングテーブルからファイル操作構造体の path に対応するファイル名を検索し，ファイル操作構造体の buff のファイル名に変更する．

このような流れでログファイルの名前変更処理が行われる．

また，ゲスト OS で管理するログファイルのファイル名と保全 OS で管理するログファイルのファイル名は，マッピングテーブルで管理している．そのため，ゲスト OS のファイル名と保全 OS のファイル名が対応していることを保証する必要がある．そこで，マッピングテーブルをファイルとして管理し，署名を行うことでマッピングテー

ブルを保証できるようにする．これでゲスト OS と保全 OS のファイル名が対応していることを保証することができるようになる．

第5章

評価・考察

本章では，実装したシステムを用いた結果とログファイルに対する攻撃への考察を述べる．

5.1 評価

提案システムの評価環境と提案システムでのログの書き込み速度を測定した．

5.1.1 評価環境

実験環境を表 5.1 に示す．

表 5.1: 実験環境

OS	:	CentOS 5.3
CPU	:	Pentium 4 (3.4GHz)
メモリ	:	512MB (保全 OS)
		384MB (ゲスト OS)
kernel	:	2.6.18

実験に用いられるシステムは，C 言語で記述されている．保全システムは，3 で述べたように保全 OS とゲスト OS を用意し，ログの書き込み速度を計測した．また，耐タンパデバイスである eToken でヒステリシス署名を行う間隔を検討し，動作させた．

表 5.2: 書き込み速度

	保全 OS のみ	提案システム
書き込み速度 (MB/s)	29.7	2.14

5.1.2 書き込み速度

ログの書き込み速度は，保全 OS のみの場合と通信機構を通じた場合を測定した．測定した結果を表 5.2 に示す．

書き込み速度の測定結果は，提案システムが保全 OS のみの場合に比べて約 10 分の 1 となった．これは，ゲスト OS でのシステムコールが呼び出される度に，通信機構を通じて保全 OS が書き込みを行っているため，システムコールの呼出し回数に比例して通信回数が多くなったためと考えられる．また，ゲスト OS，保全 OS の両 OS で書き込みを行っていることも書き込み速度が低下する要因であると考えられる．しかし，提案システムの書き込み速度でも 1 日に約 50GB のログを書き込み可能なため，保全システムの運用は可能であると考えられる．

5.2 考察

eToken を用いた際のヒステリシス署名の間隔と提案システムへの攻撃に対する考察を行った．

5.2.1 署名間隔

ログファイルの安全性を保証するために，eToken を用いた署名の処理間隔を可能な限り短くする必要がある．そのため，ログファイルのハッシュ値を計算する時間を短くしなければならない．そこで，ログファイルのハッシュ値を計算する時間を測定した．測定した結果を表 5.3 に示す．

ハッシュ値計算の測定結果から，ハッシュ値の計算は，ファイルをすべて走査する必要があるため，ファイルサイズに比例して大きくなる．3.1.1 でも述べたが，ログファイルは複数存在するため，すべてのログファイルのハッシュ値を計算するのに要する

表 5.3: ハッシュ値の計算時間

ログファイルのサイズ (MB)	ハッシュ値計算時間 (s)
1	0.020
10	0.190
20	0.377
100	1.880

時間を短くする必要がある．そのため，ハッシュ値の計算時間とログファイルの管理を考慮した結果，ログファイルのサイズを 10MB で管理することが望ましいと考えられる．システムに保存されているログファイルの数は，多く見積もっても 30 以内に抑えられるため，30 のログファイルを 10MB 単位で管理するとログファイルのハッシュ値を計算するために要する時間は，6 秒以内に抑えることができる．

また，eToken で署名を作成し，署名を eToken 内部に格納し署名した結果を返すまでの時間は 1.7 秒であった．以上のことから署名間隔は，最も短い間隔で 8 秒であることがわかる．本稿では，ログファイルの安全性と署名管理の安全性を考慮して 15 秒に設定した．15 秒に設定したのは，システムの負荷を軽減するためとシステム処理の占有率を下げるためである．

また，eToken のデータの書き換え回数が 500,000 回である [13] ことから，上記の署名間隔で約 3 年間ログファイルを保証することが可能となる．そのため，eToken の運用も実用可能であると考えられる．

5.2.2 改竄検出

ログファイルに対する攻撃に対して 4.1 で述べた署名方法で作成した署名から改竄を検出することが可能かを検討した．

ログファイルへの攻撃

ログファイルを改竄するためには，保全システムからハードディスクを取り外し，他の計算機に設置しなおすことで，ログファイルを改竄することができる．しかし，ログファイルを改竄すると，署名ファイルに記録されている署名列のハッシュ値と異な

るため，改竄を検出することになる．署名列のハッシュ値と同一のハッシュ値を持つデータを特定することはできない．そのため，ログファイルを改竄することは不可能である．

署名ファイルへの攻撃

署名した結果を記録する署名ファイルを改竄するには，署名ファイル中の最新の署名列を削除して再び署名することが考えられるが，最新の署名列を削除すると，その前の署名列の署名と耐タンパデバイスの署名が異なる値となるため，改竄を検出することができる．そのため，署名ファイルの最新の署名を削除することは不可能である．また，署名ファイル中の任意の箇所を削除したり改竄した場合，各署名列間の連鎖値が異なるため，これも不可能である．

署名間隔内での攻撃

ログファイルを改竄するには，最後に署名を行った後から次に署名を行うまでの間が可能となる．署名を作成し，次の署名を作成するまでの間は，ログファイルの正当性を保証することができないため，5.2.1 で述べた 15 秒の間はログファイルに書き込む内容を改竄することが可能となる．しかし，署名を作成し，次の署名を作成するまでの間で，write システムコールが呼び出されたときに，これからログファイルに書き込む内容が不正をしたことであるかを確認することは困難である．そのため，書き込む内容を確認した後に改竄することはより困難である．以上のことから，ログファイルや署名ファイルを改竄することが不可能であるといえる．

第6章

まとめ

ログファイルを保全するために仮想化による保全OSとヒステリシス署名を用いた保全システムを提案した．ログファイルへの書き込みに対するアクセスを追記のみに制限し，管理者が管理するゲストOSをログファイルを保全するための保全OSから分離し，ゲストOSの管理するログファイルを保全OSで管理する．ゲストOSと保全OS間には，独自の通信機構が設けてあり，この通信機構を利用することで，保全OSがゲストOSでのログファイルに対する処理を制御することができる．また，ログファイルを保全するためにヒステリシス署名を用いた．ログファイルの管理を管理者から保護することで，ログファイルに対する改竄を防止することができる．ヒステリシス署名により，複数存在するログファイルを第三者が保証することができる．これで，ログファイルや署名ファイルに対する攻撃を防止することができ，証拠性を保証することができるようになる．

また，提案システムでの書き込み速度を計測し，実用に耐えうることを確認し，安全性を向上させるための署名間隔を検討した．署名を15秒間隔で行うことができるため，この署名の間にログファイルを改竄することは難しい．また，ヒステリシス署名により作成された署名を耐タンパ領域に保管することで，ログファイルの改竄や署名ファイルの改竄を不可能にできることを確認した．

今後は，提案システムにおける書き込み速度を向上させ，システムの更なる安定化を検討する予定である．

謝辞

本研究のために、多大な御尽力を頂き、御指導を賜った名古屋工業大学の松尾啓志教授、津邑公暁准教授、齋藤彰一准教授、松井俊浩助教に深く感謝します。

また、本研究の際に多くの助言、協力をして頂いた齋藤研究室の方々に深く感謝致します。

参考文献

[1] 情報処理推進機構セキュリティセンター：
http://www.ipa.go.jp/security/index.html.

[2] 佐々木良一：日本のデジタル・フォレンジックの現状と研究動向，デジタル・フォレンジック日米共同研究ワークショップ (2006).

[3] 高田哲司，小池英樹：逃げログ：削除まで考慮にいれたログ情報保護手法，情報処理学会論文誌， Vol. 41, No. 3, pp. 823–831 (2000).

[4] Wang, Y. and Zheng, Y.: Fast and Secure Magnetic WORM Storage Systems, *Proceedings of the Second USENIX Conference on File and Storage Technologies* (2003).

[5] Wang, Y. and Zheng, Y.: Fast and Secure Append-Only Storage with Infinite Capacity, *Proceedings of the Second IEEE International Security in Storage Workshop*, pp. 11–19 (2003).

[6] Debiez, J., Hughes, J. P. and Apvrille, A.: Virtual WORM method and system, *US Patent 6615330* (2003).

[7] 芦野佑樹，佐々木良一：セキュリティデバイスとヒステリシス署名を用いたデジタルフォレンジックシステムの提案と評価，情報処理学会論文誌， Vol. 49, No. 2, pp. 999–1009 (2008).

[8] Schneier, B. and Kelsey, J.: Cryptographic Support for Secure Logs on Untrusted Machines, *The 7th USENIX Security Symposium Proceedings*, pp. 53–62 (1998).

[9] 上原哲太郎：デジタルフォレンジック：電磁的証拠の収集と分析の技術，情報処理学会誌， Vol. 48, No. 8, pp. 889–898 (2007).

[10] デジタルフォレンジック研究会：
http://www.digitalforensic.jp/wdfitm/wdf.html.

[11] 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, pp. 2381–2393 (2002).

[12] 佐々木良一，洲崎誠一：デジタル署名付文書の長期的安全性に関する考察，情報処理学会研究報告， Vol. 45, pp. 13–18 (2003).

[13] Aladdin Knowledge Systems:
http://www.aladdin.com/.

[14] RSA Laboratories:
http://www.rsa.com/rsalabs/node.asp?id=2133.

[15] Xen:
http://www.xen.org.