修士論文

動的なインフォメーションフロー制御による情報漏洩防止手法

Dynamic Information Flow Control for Preventing Information Leakage

平成19年2月2日提出

指導教員　坂井 修一 教授

東京大学大学院 情報理工学系研究科
電子情報学専攻

栗田　弘之

概要

コンピュータシステムが扱う情報は多様化しており，個人情報や企業の機密情報，商用マルチメディアコンテンツなどにおいては，それらの漏洩が社会問題となっている．インターネットが発達した現在では，漏洩した情報は短期間に広く流通し，その回収もきわめて困難であるため，情報漏洩の被害は甚大なものとなる．

コンピュータシステムから情報が漏洩する原因は，ユーザの悪意や過失をはじめとして，スパイウェアやトロイの木馬型プログラムといった悪意あるプログラム，脆弱性を持ったプログラムに対する外部からの攻撃など，多岐にわたる．

従来の情報漏洩対策では，個々の情報漏洩の原因に応じていくつかの手法が用いられている．主な手法としては，OS によるアクセス制御と Windows Media DRM のようなディジタル著作権管理システムがあげられる．しかし，これらの手法はプログラマの悪意やプログラムの脆弱性の有無，耐タンパ性といったプログラムの信頼性に依拠している．利用するプログラムの信頼性をユーザが判断することや，プログラマがプログラムの信頼性を第三者に保証することは一般に困難であり，従来の情報漏洩対策手法は十分な機能を提供しているといえない．

本論文では，プログラム実行時の情報の流れ (インフォメーションフロー) に着目し，プログラムの信頼性に依存しない情報漏洩防止手法を提案する．提案手法では，プログラム実行時にプロセッサと OS がインフォメーションフローを追跡，制御することにより，保護するべきデータがプログラム外部に不用意に出力されることを防止する．インフォメーションフローは大きく，データフローに付随する明示的フローと，コントロールフローに付随する暗黙的フローに分けられる．提案手法では，明示的フローは，保護するべきデータを識別するタグ (出力制限タグ) を命令実行時に伝搬させることで追跡し，暗黙的フローは，保護観察モードと呼ぶ新たな実行モードの導入によって制御する．

提案手法は情報漏洩防止の機能をプロセッサと OS が提供し，その強度はプログラマの悪意やプログラムの脆弱性の有無，耐タンパ性といったプログラムの信頼性に依拠しない．また，保護するべき情報がプログラム外部に出力される際には，著作権者などが定める保護ポリシに従ってその可否を判断するため，ユーザの悪意や過失にも依拠しない．加えて，システムコールの形でプログラマへのインタフェースを提供することで，プログラム本来の多様な機能が過剰に制限されることを防止する．

システムレベルエミュレータ Bochs を用いて提案手法の実装を行った．Bochs はシステムレベルエミュレータであり，プロセッサ，メモリに加えて，グラフィック，

ネットワーク，ディスクドライブ等のデバイスをエミュレートする．今回は，提案手法を実装した Bochs 上でいくつかのプログラムを実行し，実装システムの挙動を確認した．また，メモリ消費量とパフォーマンスに対するオーバーヘッドを定量的ないし定性的に評価した．

目 次

図目次

表目次

第1章　はじめに

1.1　背景

情報機器の発達とその低廉化に伴って，コンピュータシステムが扱う情報は多様化しており，ありとあらゆる情報がディジタルデータとして活用，保存されている．また，インターネットの広帯域化によって，ディジタルデータのやりとりが容易になり，ソフトウェアや音楽，映像などのコンテンツがオンラインで提供されるようになっている．しかし，情報のディジタル化はこうした利便性を生む一方で，情報漏洩という大きな弊害を生じさせている．

個人情報や企業の機密情報，商用マルチメディアコンテンツなどにおいては，それらの漏洩が社会問題となっている．漏洩した情報はインターネットによって短期間に広く流通し，その回収は困難であるため，情報漏洩の被害は甚大なものとなる．

コンピュータシステムから情報が漏洩する原因は，ユーザの悪意や過失をはじめとして，スパイウェアやトロイの木馬型プログラムといった悪意あるプログラム，脆弱性を持ったプログラムに対する外部からの攻撃など，多岐にわたる．

従来の情報漏洩対策では，個々の情報漏洩の原因に応じていくつかの手法が用いられている．主な手法として，OS によるアクセス制御と Windows Media DRM のようなディジタル著作権管理システムがあげられる．

OS によるアクセス制御では，ユーザやプログラム毎にファイルなど各種リソースへのアクセスを制御する．これにより権限のないユーザによる不正なリソースアクセスを防止できる．しかし一方で，あるリソースへのアクセスを許可されたプログラムが，そのリソースから読み込んだデータをどう利用し，どこに出力するかに関しては制御できない．そのため，プログラム作成者が悪意を持ってデータを漏洩させようとする場合や，脆弱性を持ったプログラムが外部からの攻撃によって乗っ取られた場合などに，ユーザの意志に反してデータを漏洩してしまう．

ディジタル著作権管理システム [7,8] では，音楽データなどのコンテンツが暗号化された状態で配布され，Windows Media Player のような専用プログラムがそれを復号化し，コンテンツを再生する [9,17]．一般にディジタル著作権管理システムでは，コンテンツを復号可能なプログラムを専用プログラムに限定し，その専用プログラムの信頼性を高めることで，コンテンツを保護している．しかし，復号化処理をプログラム内部で行っているため，コンテンツ保護の強度はプログラムの耐タンパ性に依拠する [6]．プログラムの耐タンパ性は暗号強度に比べ，その

向上が困難であり，リバースエンジニアリング等の手段によってコンテンツが漏洩する恐れがある．また，コンテンツを復号可能なプログラムが専用プログラムに限定されることは，ユーザの利便性を損なう要因となる．

このように，既存の情報漏洩対策技術は，プログラム作成者の悪意や脆弱性の有無，耐タンパ性といったプログラムの信頼性を前提としている．しかし，ユーザ環境で利用されるプログラムは多岐にわたっており，プログラム作成者がプログラムの信頼性を保証したり，ユーザがプログラムの信頼性を判断することは一般に極めて困難である．

1.2 目的

本研究は，プログラムの信頼性に依存しない情報漏洩防止の実現を目的とする．また，情報漏洩を防止するに当たり，プログラムの機能が過剰に制限されることを防ぎ，ユーザの利便性と情報漏洩の防止を両立する．

本研究では，プログラムの信頼性に依存しない情報漏洩防止機能の実現のため，プログラム実行時の情報の流れ (インフォメーションフロー) に着目する．プログラム実行時にプロセッサと OS がインフォメーションフローを追跡，制御することで，保護するべきデータがプログラム外部に不用意に出力されることを防止する．

提案手法では情報漏洩防止の機能をプロセッサと OS が提供するため，その強度はプログラマの悪意やプログラムの脆弱性の有無，耐タンパ性といったプログラムの性質に依拠しない．また，保護するべき情報がプログラム外部に出力される際には，著作権者などが定める保護ポリシに従ってその可否を判断する．そのため，システムを利用するユーザの悪意や過失にも依拠しない．加えて，システムコールの形でプログラマへのインタフェースを提供することで，プログラム本来の多様な機能が過剰に制限されることを防止する．

1.3 構成

本論文は，以下のように構成される．

2 章で情報漏洩の防止に向けた本研究がとるアプローチである情報漏洩防止プラットフォームについて説明する．プラットフォームが情報漏洩を防止するメカニズムや，その構成，実現に向けた要件について述べる．

3 章では，インフォメーションフローについて説明する．インフォメーションフローは大きく，データフローに付随する明示的フローと，コントロールフローに付随する暗黙的フローに分けられるが，それらの性質についてここで述べる．

4 章では提案手法の詳細について述べる．提案手法は大きく明示的フローの追跡に関する部分と，暗黙的フローの制御に関する部分に分けられる．明示的フローは，保護するべきデータを識別するタグ (出力制限タグ) を命令実行時に伝搬させ

ることで追跡し，暗黙的フローは，保護観察モードと呼ぶ新たな実行モードの導入によって制御する．

5 章で提案手法の実装，評価について述べる．6 章で関連研究について説明した後，最後に 7 章でまとめと今後の課題について述べる．

第2章　情報漏洩防止に向けたアプローチ

2.1　情報漏洩防止プラットフォーム

情報漏洩とは，情報の保有者や情報に関して権利をもつ者の意図に反して，第三者に情報が渡ることである．コンピュータシステムにおいては，機密データがシステム外部に出力されることによって情報漏洩が発生する．

情報漏洩防止に向けて本研究がとるアプローチの概念図を図 2.1 に示す．

プログラムは一般に，入力されたデータに対して定められた処理を行い，プログラム外部に何らかの出力を行う．たとえば音楽再生プログラムであれば，エンコードされた音楽データを入力として読み込み，ヘッダの解析，デコードなどの処理を行った後，システムコールを介してサウンドデバイスのデバイスドライバに出力する．

ここで，プログラム実行環境がプログラム実行時の情報の流れ (以下，インフォメーションフロー) を動的に追跡ないし制御できるならば，プログラムの出力に注目して情報漏洩を防止することが可能である．それは，インフォメーションフローの追跡・制御によって，プログラムが出力しようとするデータと，プログラムに入力されたデータとの依存関係がわかり，プログラム実行環境が出力の正当性を検証できるためである．この検証によって，保護するべきデータが不用意にプログラム外部に出力されることを防止できる．

本研究ではこのようなプログラム実行環境を情報漏洩防止プラットフォームと呼び，その実現を目的とする．情報漏洩防止プラットフォームはプロセッサおよび OS から構成される．

情報漏洩防止プラットフォームを用いた場合，先に述べた音楽再生プログラムの例であれば，プログラムが音楽データを不正にネットワークやディスクに出力しようとした場合に，それを検出することが可能になる．そのため，悪意を持った音楽再生プログラムが，別の目的で通信を行う振りをしながら，実際には音楽データをネットワークに出力するといった形の情報漏洩を防止できる．

情報漏洩防止プラットフォームの特徴を以下に示す．

- アクセス制限などの既存技術とは異なり，プログラムの信頼性を前提としない．

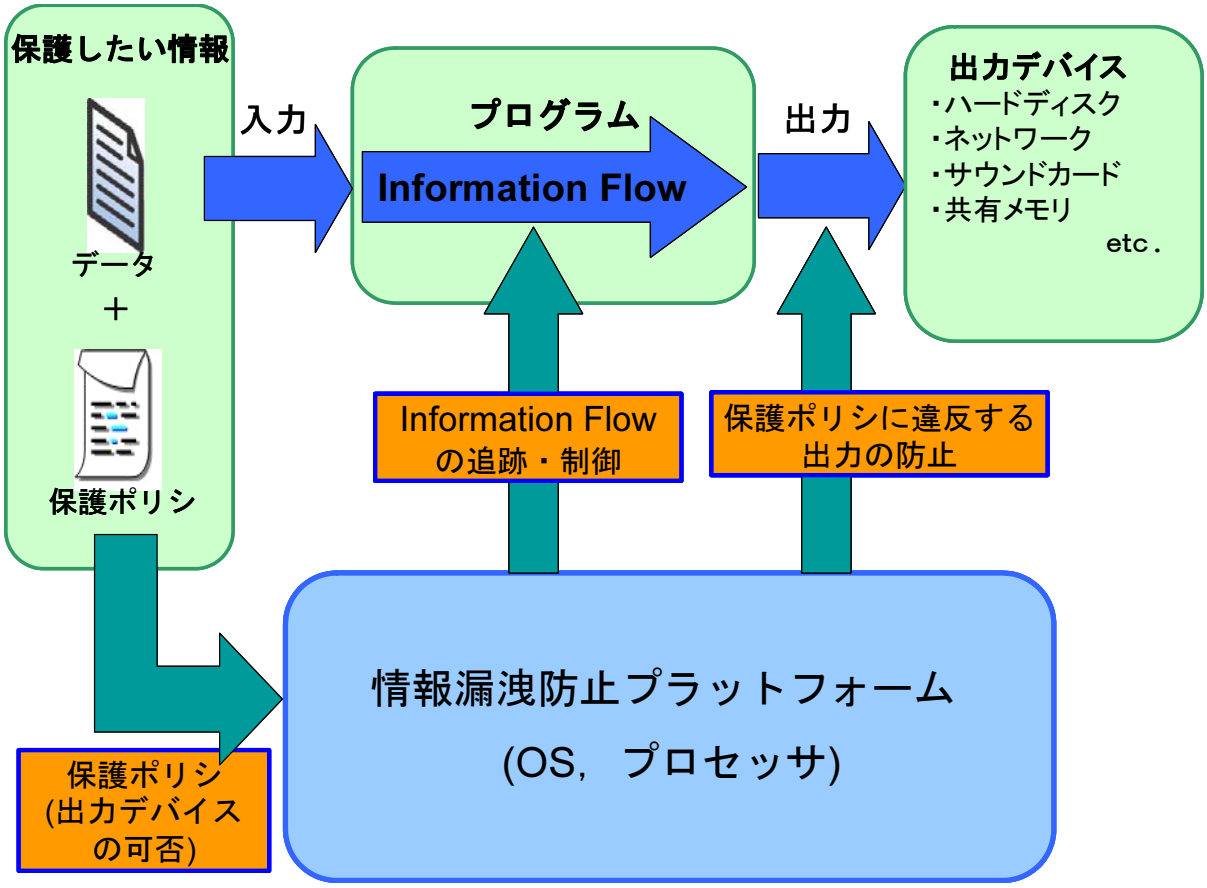

図 2.1: 情報漏洩防止プラットフォーム概念図

- トロイの木馬のようなウイルスやスパイウェアを含め，システム上で動作するすべてのプログラムの実行において有効である．
- ファイルからの入力だけでなく，ユーザのキー入力やネットワークからのストリーミングデータなど，プログラムに入力されるあらゆるデータが保護の対象となる．

カバートチャネル

プログラムからの情報伝達は必ずしもデータの出力によってのみ起こるわけではない．コンピュータシステムから外部に情報を伝える経路を一般にチャネルと呼び，本来の目的が情報伝達ではないチャネル，つまりデータ出力以外のチャネルを一般にカバートチャネル (隠れチャネル) と呼ぶ．代表的なカバートチャネルを以下に示す．

- Termination Channel

  プログラムの実行を終了するかしないかによって情報を伝えるチャネル．たとえば特定の入力に対して実行時例外が発生しプログラムが終了する場合，終了したという事象から入力に関する情報が漏洩する．

- Timing Channel

  プログラムが特定のアクションを起こすタイミングによって情報を伝えるチャネル．アクションの例としてプログラムの終了があり，プログラムの合計実行時間などによって情報が漏洩する．インフォメーションフローセキュリティからは外れるが，RSA の秘密鍵を暗号処理時間から推測する場合なども Timing Channel に含まれる．

- Resource Exhaustion Channel

  メモリやディスクなど，複数のプログラムで共有される有限なリソースの消費状況によって情報を伝えるチャネル．

- Power Channel

  電力消費量によって情報を伝えるチャネル．

カバートチャネルによる情報漏洩は，情報を不正に入手しようとする攻撃者がどの事象を観測できるかに依存する．攻撃者がプログラムの終了を検知できるならば Termination Channel が問題となり，さらに開始も検知できる場合には Timing Channel が問題となる．また，電力消費量を測定できる場合には Power Channel が問題となる．

プログラミング言語ベースのインフォメーションフロー解析では，”コンパイル時に検査されない例外 (unchecked exceptions) ”の存在を認めないといったものをはじめとして，カバートチャネルの克服を目指した研究がなされているが，カバートチャネルによる情報漏洩を完全に防止することはできていない [12]．本研究においても，これらカバートチャネルによる情報漏洩は直接的には範囲外とする．

2.2 情報漏洩防止プラットフォームの実現に向けた要件

本論文では次章以降，情報漏洩防止プラットフォームの実現に向けた要件のうち，基本的にはインフォメーションフローの追跡・制御に議論を限定するが，ここでは，それ以外の要件について簡単に述べる．

2.2.1 保護ポリシ

情報漏洩防止プラットフォームにおいて，あるデータを情報漏洩から保護したい場合，そのデータに関して，どのデバイスへの出力を認め，どのデバイスへの出力を禁止するかをあらかじめ決めておく必要がある．

こうした出力先デバイスの可否情報を，本稿では保護ポリシと呼ぶ．保護ポリシの例としては,「ネットワークへの出力の禁止」や,「認証されたサウンドデバイスのデバイスドライバへの出力のみ許可」,「暗号化した状態でのディスクへの保存は許可」などが考えられる．保護ポリシはデータの著作権者などが記述することを想定している．また，保護ポリシは電子署名等によって完全性が保証されるものとする．

保護したいデータとその保護ポリシは対応付けられた状態で保存され，保護したいデータに依存したデータがプログラム外部に出力される際には，保護ポリシを基にその可否を判断する．ディジタル著作権管理の分野では，より汎用な保護ポリシの記述に関する研究が行われている [2, 4, 15] が，本研究では保護ポリシの細かい仕様に関しては範囲外とする．

2.2.2 OS およびデバイスドライバの役割とその完全性

プログラムの出力は，基本的に OS が提供するシステムコールを介して行われるため，出力の可否は OS が判断する．また詳細は後述するが，インフォメーションフローの動的な追跡・制御においても OS は重要な役割を果たす．そのため，OS が悪意を持って改竄された場合，情報漏洩の防止を保証することはできない．本研究では，TPM [3] などのハードウェアを起点とするセキュリティ技術によって OS の完全性が保証されることを仮定する．

また，プログラムの出力は最終的に出力デバイスのデバイスドライバに委ねら

れる．そのためデバイスドライバについてもその正当性と完全性が保証される必要がある．本研究では，電子署名等の手段によってこれらが満たされるものとする．

第3章　インフォメーションフロー

インフォメーションフローは文字通りプログラム実行時の情報の流れを意味する．インフォメーションフローは大きく，データフローに付随する明示的フローと，コントロールフローに由来する暗黙的フローに分類される．本節では，これら二つのフローおよびカバートチャネルについてについて説明すると共に，動的な暗黙的フロー追跡の困難性について述べる．

3.1　明示的フロー

プログラムの実行においては，数値演算や論理演算などで多くの演算が行われる．たとえば，

$$y = x + 5$$

という演算では，x と 5 の和を y に代入する．このとき，演算後の y の値から x を決定できるため，この演算によって明らかに x に関する情報が y に伝わっている．このようにデータフローに付随する形で情報が伝わるフローを明示的インフォメーションフロー (以下，明示的フロー) と呼ぶ．

明示的フローによって伝わる情報には，不完全な情報も含まれる．たとえば，

$$y = x\%2$$

という演算では，演算後の y の値から x の値を決定することはできない．しかし，x に関する情報の一部が y に伝わっており，このような場合にも x から y への明示的フローが存在する．

次に，

$$y = *x$$

のようなポインタ変数を介した代入を考える．この代入では，y にアドレス x の示す値 (z とする) が代入される．このとき値がコピーされる z から y だけでなく，アドレス x から y に対する明示的フローも存在する．これは z の値から z のアドレスである x を決定できる場合 (たとえば，$z = x$ のような時)，代入後には y からも x が決定可能になるためである．

明示的フローはデータフローに付随して発生するため，データフロー同様，命令セットアーキテクチャの一命令の実行において完結する．

```
if(x == true)
    y = true;
else
    y = false;
```

図 3.1: x → y の暗黙的フローが発生するコード

3.2 暗黙的フロー

図 3.1 に示したコードの実行を考える．x がブール変数であれば，このコードが実行すると y の値は x と一致する．そのため，明らかに x から y へのインフォメーションフローが存在する．しかし，3.1 で述べた明示的フローとは異なり，x と y の間に明示的な代入・演算関係はない．

この例のように，条件分岐を基点とし，分岐後のパスで行う処理によって発生するインフォメーションフローを暗黙的インフォメーションフロー (以下，暗黙的フロー) と呼ぶ．暗黙的フローはコントロールフローに由来するインフォメーションフローであるといえる．暗黙的フローは，条件分岐で条件として使われた値 (先の例では x) とその条件分岐に依存した処理のデスティネーション (先の例では y) の間に存在する．なお，ここでの処理とは具体的には，メモリやレジスタの書き換えをさす．

3.3 動的な暗黙的フロー追跡の困難性

図 3.1 のコードと同様，図 3.2 に示したコードにも x から y への暗黙的フローが存在し，コードの実行後には x と y の値は一致する．ここでこの暗黙的フローを，条件分岐後に行う処理に注目することで，動的に追跡することを考える．

図 3.3 に図 3.2 のコードのコントロールフローグラフを示す．図 3.3 に示すとおり，コード実行時の挙動は以下のようになる．

- x が true のとき，条件分岐後，y への代入を行う．
- x が false のとき，条件分岐後，いかなる処理も行わない．

このため，条件分岐後に行う処理に注目しても，x が false のときには，x から y への暗黙的フローを追跡できない．

3.2 で述べたように，暗黙的フローは分岐後のパスで行う処理の違いによって発生するため，条件分岐後のパスで「何も処理を行わない」場合でも暗黙的フロー

```
y = false;
if(x == true)
    y = true;
```

図 3.2: x → y の暗黙的フローが発生するコード (2)

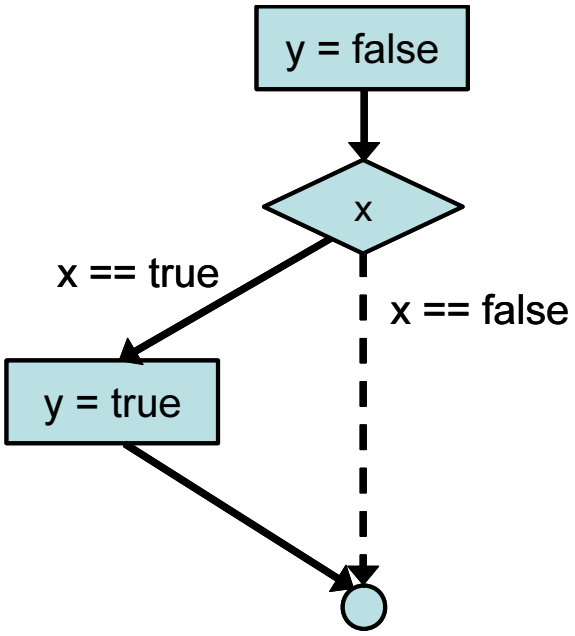

図 3.3: 図 3.2 のコードのコントロールフローグラフ

は発生する．一般にすべての暗黙的フローを認識するためには，条件分岐後にとり得るすべてのパスに関して，そのパスがどのような処理を行うかを解析する必要がある．つまり，プログラムが実行時に通過するただ一つのパスの観測では，すべての暗黙的フローを認識することはできない．

次章では，暗黙的フローを追跡するのではなく，暗黙的フローの発生期間に注目した制御手法について述べる．

第4章　提案手法

前章でのインフォメーションフローに関する考察をふまえ，本章では，以下のようなインフォメーションフローの追跡・制御手法を提案する．

- 保護するべき情報を識別するタグ (出力制限タグ) を命令実行時に伝搬させることで，明示的フローを追跡する．
- 保護観察モードと呼ぶプログラムの新たな実行モードを導入することで，暗黙的フローを制御する．
- プログラマに対して保護観察モードに関するインタフェースを提供し，プログラム機能の柔軟性が損なわれることを防止する．

以下，まず 4.1 で明示的フローの追跡について述べ，4.2 以降で暗黙的フローの制御について述べる．

4.1　明示的フローの追跡

4.1.1　出力制限タグ

メモリやレジスタに格納されたデータがどの保護ポリシによる出力制限を受けるかを識別するために，メモリやレジスタを拡張し，出力制限タグを付加する (図 4.1).

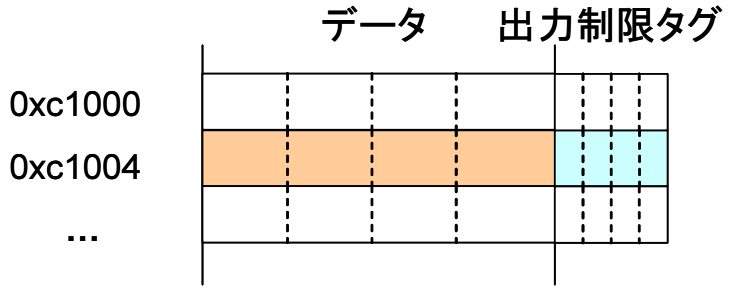

図 4.1: 出力制限タグ

表 4.1: 各命令実行時の出力制限タグ伝搬

命令	命令内容	出力制限タグ伝搬
ADD r1, r2, r3	r1 = r2 + r3	$\underline{r1}$ = $\underline{r2}$ OR $\underline{r3}$
AND r1, r2, r3	r1 = r2 & r3	$\underline{r1}$ = $\underline{r2}$ OR $\underline{r3}$
LOAD r1, r2	r1 = [r2]	$\underline{r1}$ = $\underline{[r2]}$ OR $\underline{r2}$
STORE r1, r2	[r2] = r1	$\underline{r2}$ = $\underline{[r1]}$ OR $\underline{r1}$

出力制限タグの初期化はプログラムがリソースからデータを読み込む際に，OS が以下のように行う．なお，タグの内容をプログラムが自由に変更することはできない．

- 読み込み元のリソースにいずれかの保護ポリシが設定されている場合，読み込んだデータの出力制限タグにその保護ポリシを示す ID をセットする．
- リソースに保護ポリシが設定されていない場合，出力制限タグとして 0 をセットする．

以下，出力制限タグがセットされたデータを出力制限データ，それ以外のデータを非出力制限データと呼ぶ．なお，非出力制限データの出力制限タグは 0 とする．

メモリ上のデータの出力制限タグは，ページテーブルに似た階層構造のタグテーブルによって，データとは独立に保存，管理することを想定しており，その詳細については次章で述べる．

4.1.2 出力制限タグ伝搬

3.1 で述べたように明示的フローはデータフローに付随して発生する．そのため一つの明示的フローは，データフロー同様に命令セットアーキテクチャ(ISA) における一命令の実行において完結する．そこで各命令の実行時に，命令のソースからディスティネーションに対して出力制限タグを伝搬させることで，明示的フローの追跡が可能である．出力制限タグの伝搬は論理和 (OR) 演算によって行い，ソースの中に一つでも出力制限データがあれば，ディスティネーションは出力制限データとなる (図 4.2).

代表的な RISC 命令を例に，タグ伝搬の様子を表 4.1 に示す．ただし，$\underline{r1}$ はレジスタ r1 の出力制限タグを表し，[r1] はアドレス r1 が示すメモリの値を表す．3.1 で述べたように，LOAD，STORE 命令ではアドレスからも出力制限タグが伝搬する．

出力制限データがシステムコールを介してプログラム外部に出力される際，その出力が出力制限タグの示す保護ポリシの内容に違反しないかどうかを OS が判断する．これによって明示的フローを介した情報漏洩が防止される．

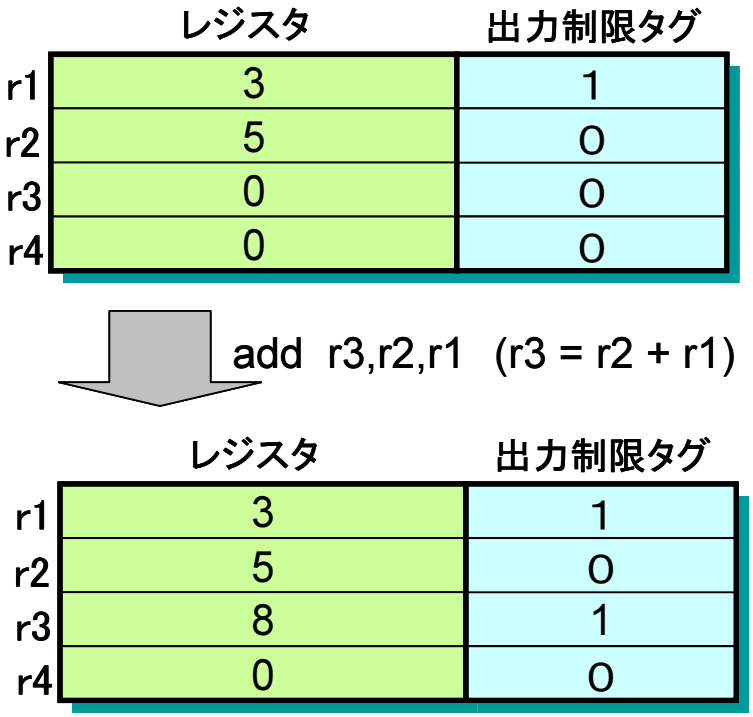

図 4.2: 出力制限タグの伝搬

出力制限タグのサイズと最大保護ポリシ数

出力制限タグのサイズ (ビット数) は，プログラムが同時に扱うことのできる保護ポリシの数 (最大保護ポリシ数) を決定する．しかし，メモリ上に保存されるデータの出力制限タグはメモリ上で保存，管理するため，タグサイズの増大はメモリ容量を圧迫する．そのため，むやみに出力制限タグを大きくすることはできない．

出力制限タグのビット数と最大保護ポリシ数の関係について考える．先に述べたように，出力制限データ同士を演算した場合，デスティネーションは当然，出力制限データとなる．二つの異なる出力制限タグを持ったデータ同士を演算した場合，デスティネーションはその二つの出力制限タグが示す保護ポリシを同時に満たさなければならない．これは，デスティネーションの出力制限タグに格納する新たな保護ポリシ ID が必要になることを意味する．この新たな保護ポリシ ID を既存の保護ポリシ ID の論理和によって表現するためには，出力制限タグの各ビットがそれぞれ固有の保護ポリシを表す必要がある．この場合には出力制限タグのビット数がそのまま最大保護ポリシ数となる．

別の方法としては，新たな保護ポリシ ID が必要になる演算 (0 ではない二つの異なる出力制限タグをもつデータを用いた演算) の実行時に，プロセッサが例外を発生させ，OS に新たな保護ポリシ ID を決定してもらう可能性がある．この場合，出力制限タグが 4 ビットであれば，最大保護ポリシ数は $2^4 - 1 = 15$ となる．この

方法は，実行時のオーバーヘッドが大きいため，新たな保護ポリシ ID が必要となる命令が少ない場合のみ現実的である．

本研究では，最大保護ポリシ数は実用上それほど大きい必要はないと考えている．それは，一つのプログラムが扱うデータにはプログラム毎に固有の特徴があり，保護ポリシが異なるデータ (デバイスレベルで出力制限の可否が異なるデータ) を同時に複数扱う状況は限定されると考えるためである．また同様に，複数の異なる保護ポリシを使った演算 (新たな保護ポリシ ID が必要になる演算) が発生する状況もきわめて限定的であると考える．以上の理由から，32 ビットのデータをタグ付けの最小単位とし，それに対して 8 ビットの出力制限タグを付加することを想定している．

4.2 暗黙的フローを介した情報漏洩の防止に向けたアプローチ

3.3 で述べたように，暗黙的フローをプログラム実行時に正確に追跡することはできない．しかし，暗黙的フローの発生は，条件分岐を行ってから，その分岐が合流するまでの期間に限定される．これは，暗黙的フローの発生が分岐による実行パスの違いに起因するためであり，分岐の合流後は実行パスが同一になるため暗黙的フローが発生しない．そこで，暗黙的フローを介した情報漏洩を以下のような方針で防止する．

- 出力制限データを起点とする暗黙的フローが発生する期間を区別し，その間の実行モードを保護観察モードとする．
- 保護観察モード中のプログラム外部とのインタラクション，特にデータ出力を保護ポリシに基づいて制限することで，保護観察モード中の情報漏洩を防止する．詳細は 4.4 で述べる．
- 保護観察モード中に発生した暗黙的フローの影響が保護観察モード終了後に及ばないように制御することで，保護観察モード終了後の情報漏洩を防止する．詳細は 4.5 で述べる．

4.3 保護観察モードへの移行および終了

4.3.1 保護観察モードへの移行

保護観察モードは，出力制限データを起点とする暗黙的フローが発生する期間の実行モードである．そのため保護観察モードへの移行条件は，暗黙的フローの発生条件と同じく，出力制限データに依存した条件分岐命令ないし制御移行命令

表 4.2: 保護観察モードへの移行条件

命令	命令内容	移行条件
BRANCH r1, offset	if(r1) PC = PC + offset	r1 != 0
JMP r1	PC = r1	r1 != 0

出力制限タグ	終了ポイント

図 4.3: 保護観察モードステータス

の実行である．代表的な RISC 命令を例に，保護観察モードへの移行条件を表 4.2 に示す．ただし，r1 はレジスタ r1 の出力制限タグを表し，PC はプログラムカウンタを表す．

保護観察モードへ移行するとき，プロセッサ内部の保護観察モードステータス (図 4.3) に分岐に使用した出力制限タグの値 (表 4.2 の例であれば，r1) を格納する．保護観察モードステータスのタグの値は，その保護観察モードがどの保護ポリシによる制限を受けるものであるかを示す．保護観察モード中の制限については後述する．

マルチスレッドプログラムにおいて保護観察モードへの移行はスレッド単位で行われる．そのため保護観察モードステータスはスレッドコンテキストの一部であり，スレッド切り替え時に保護観察モードステータスも同時に切り替える．

4.3.2 保護観察モードの終了

保護観察モードは保護観察モードに移行した際の分岐が合流した時に終了する．しかし，一般的な命令セットアーキテクチャでは特定の分岐の合流を動的に検出することはできない．そこで，プログラマに特定の命令アドレスを，保護観察モードの終了アドレスとして申告してもらうことを考える．終了アドレスの申告は通常モード中に，システムコールを介して行われ，保護観察モードステータスに格納される．保護観察モードはプログラムカウンタが終了アドレスと一致した時点で終了し，通常モードへと移行する．

保護観察モードの終了アドレスはプログラマが自己申告するものであるが，プログラマが悪意をもって虚偽の命令アドレスを申告したとしても問題は発生しない．それは，保護観察モードが以下の三つの性質をもつためである．

- 保護観察モード中の終了アドレスの申告を禁止する．
- 保護観察モード時にどのような実行パスを通ったとしても，プログラムカウ

```
If (x == true)
    printf("1");
else
    printf("0");
```

図 4.4: 非出力制限データの出力による情報漏洩の例

ンタが終了アドレスに到達するまで保護観察モードは続行される．

- 保護観察モード終了後は，それまでの実行パスにかかわらず，同じ命令列が実行される．

このことからもわかるように，プログラマが申告する終了アドレスは正確に分岐の合流ポイントである必要はなく，合流ポイント以降の任意の命令アドレスでかまわない．また，後に述べる保護観察モード中の制限がプログラム機能の実現上問題にならない場合には，必ずしも保護観察モードを終了する必要はない．その場合にはプログラムが実行を終了するまで，保護観察モードとなる．

4.4 システムコール呼び出しの制限

保護観察モードは，出力制限データに依存した分岐によって移行するため，保護観察モード中は実行パスそのものが出力制限データに依存している．そのため，保護観察モード中のプログラム外部とのインタラクションは，出力制限データに関する情報の漏洩につながる．特に，プログラム外部へのデータ出力は，それが非出力制限データであったとしても，出力されたデータから実行パスが特定されることによって，出力制限データの情報が漏洩する恐れがある．

保護観察モード中の非出力制限データの出力によって出力制限データ (x) の情報が漏洩する例を図 4.4 に示す．このコードの実行時に標準出力を監視していれば，1 が出力されたとき，x が true であることがわかり，0 であれば，x が false であることがわかる．

データ出力を含む，プログラム外部とのインタラクションはシステムコールを介して行われる．そこで，保護観察モード中のシステムコール呼び出しを保護ポリシに基づいて制限することで，保護観察モード中の情報漏洩を防止する．この制限は，保護観察モードステータスの出力制限タグの値を元に，OS が個々のシステムコールの可否を判断することで行う．

表 4.3: システムコールの呼び出し制限

システムコール名	システムコール内容	システムコールの制限条件
write	ファイルディスクリプタへの出力	平文でのディスクへの出力が禁止されている場合
send	ネットワークソケットへの出力	ネットワークへの出力が禁止されている場合
ioctl	デバイスの制御	該当するデバイスへの出力が禁止されている場合
mkdir	ディレクトリの作成	平文でのディスクへの出力が禁止されている場合

Linux のいくつかのシステムコールを例に，システムコール呼び出しの制限条件を表 4.3 に示す．たとえば，保護ポリシがネットワークへの出力を禁止するものであった場合，保護観察モード中はすべての send が禁止される．同様に，特定のデバイスドライバへの出力のみ許可する保護ポリシであった場合には，そのデバイスドライバへの ioctl 以外，すべての write, send, mkdir 等が禁止される．

4.5 暗黙的フローの制御

保護観察モード中は，出力制限データを起点とする以下の二種類の暗黙的フローが発生する．

- デスティネーションが出力制限データである暗黙的フロー
- デスティネーションが非出力制限データである暗黙的フロー

出力制限データは通常モード中 (保護観察モード終了後) も出力制限を受けるため，前者の暗黙的フローは問題にならない．一方，後者の暗黙的フローは，出力制限データの情報が非出力制限データに伝わるため，保護観察モード終了後に出力制限データに関する情報が漏れる原因となる．

この問題に対して，単純には，通常モード時の出力制限タグ伝搬と同様に，保護観察モード中に実行する命令のデスティネーションに対して出力制限タグを付加すればよいように思われる．しかし，この方法がうまくいかないことを，図 4.5 に示したコードを用いて説明する．このコードには，$x \rightarrow y$ および $y \rightarrow z$ の暗黙的フローが存在し，結果として x と z は一致する．図 4.5 のコードのコントロールフローグラフを図 4.6 に示す．ただし，実線は x が true の時の実行パスを表し，破線は x が false の時の実行パスを表す．

```
1:  y = true;
2:  z = true;
3:  if( x == true )
4:      y = false;
5:  if( y == true )
6:      z = false;
```

図 4.5: x → y および y → z の暗黙的フローが存在するコード

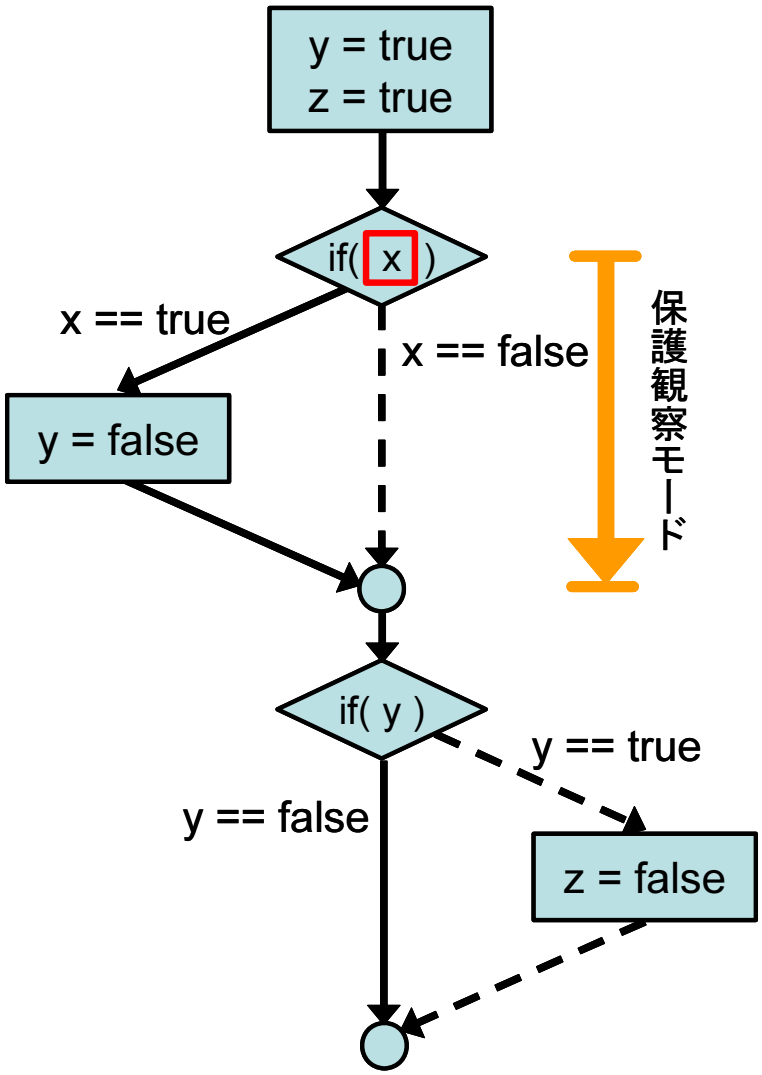

図 4.6: 図 4.5 のコードのコントロールフローグラフ

コード開始時点で，x が出力制限データ，z と y は非出力制限データであるとする．コードを実行するとまず，3 行目の if 文によって保護観察モードへ移行し，その後の挙動は x の値に応じて以下のようになる．なお，図 4.5 のコードでは省略しているが，保護観察モードは，プログラマによる終了アドレスの申告によって，分岐の合流ポイントで終了するものとする．

- x が true の場合，4 行目の代入によって y に出力制限タグが付加される．3 行目の if 文による分岐はここで合流するため，一旦通常モードに戻るが，y に出力制限タグが付加されたため，5 行目の if 文によって再び保護観察モードへと移行する．しかし，このとき y は false であるため，6 行目の代入は実行されず，z には出力制限タグが付加されない．

- x が false の場合，4 行目の代入は実行されずに保護観察モードは終了する．このとき y には出力制限タグが付加されないため，5 行目の if 文では保護観察モードに移行しない．そのため 6 行目の代入が実行されても，z には出力制限タグが付加されない．

このように，いずれの場合も z には出力制限タグが付加されない．これは，3.3 で述べた暗黙的フロー追跡の困難性に起因する本質的な問題である．

4.5.1 保護観察違反検出方式

出力制限タグを拡張して保護観察ビットを付加し，以下のような保護観察違反検出方式を考える．

- 保護観察モード中に非出力制限データを変更した場合，保護観察ビットを立てる．

- 通常モード時に保護観察ビットの立ったデータを使用した場合，保護観察違反としてプログラムの実行を停止する．

- 保護観察ビットは通常モード時にデータが上書きされた時に下げる．

保護観察ビットの導入によって，保護観察モード中に変更した非出力制限データ，すなわち暗黙的フローの終点となった非出力制限データを，通常モード時に読むことができなくなる．これによって，保護観察モード中に発生した暗黙的フローの影響が保護観察モード終了後に及ぶことを防止する．

なお，マルチスレッドプログラムでは，通常モードのスレッドと保護観察モードのスレッドが同時に存在しうるが，通常モードスレッドが保護観察ビットの立ったデータにアクセスした時点で保護観察違反となり，プログラムは実行を停止する．

```
y = 0;
for( i = 0; i < 32; i++){
    mask = 1 <<  i;
    if( x & mask ) y = 1;
    printf("%d", y);
}
```

図 4.7: 出力制限データ x を 1 ビットずつ漏洩させるコード

4.5.2 保護観察違反検出方式の有効性

図 4.6 の例において保護観察違反方式を用いた場合，以下のような挙動となる．

- x が true の場合，4 行目の代入によって y に保護観察ビットが付加される．ここで 3 行目の分岐は合流し，通常モードに戻る．5 行目の if 文において保護観察ビットのついた y を使用するため，保護観察違反が発生する．
- x が false の場合，4 行目の代入は実行されず，y には保護観察ビットが付加されない．そのため，5 行目の if 文で保護観察違反は発生しない．

このように，x が true の場合は保護観察違反を検出しプログラムは実行を停止するが，x が false の場合には，$z = x$ のまま実行を続行する．z は非出力制限データであるため，これは z を介して x の情報が漏洩する恐れがあることを意味する．

しかし，この問題があっても本手法はなお有効である．この問題を利用して，32bit の出力制限データを外部に漏洩する場合について考える．

まず，図 4.7 に示すコードによって，出力制限データ x を 1 ビットずつ漏洩させることを考える．x が 32 ビットの値を等確率でとるとすれば，1 ビットの試行 (printf の実行) のたびに，$\frac{1}{2}$ の確率で保護観察違反が発生する．そのため，保護観察違反でプログラムが停止する前に漏洩できる平均ビット数は，以下の式が示すようにわずか 2 ビットである．

$$\lim_{n\to\infty}\sum_{k=1}^{n}(\frac{1}{2}^{k})k = 2$$

一方，図 4.8 に示すコードを実行した場合，標準出力に出力される 0 の数をカウントすることで，x の値を決定できる．しかし，このコードの実行には，漏らしたいビット数 (今回の場合は 32) に対して幾何級数的な時間および出力サイズを必要とする (今回の場合は 2^{32})．そのため，現実的な時間で多くのビット数のデータが漏れる恐れはなく，実用上の問題は小さい．また，保護観察違反によって停止したプログラムは以降保護ポリシの設定されたファイルにアクセスできない，といったアクセス制限との組み合わせも有効である．

```
y = 0;
for( i = 0; i <= 0xffffffff; i++){
    if( x == i ) y = 1;
    printf("%d", y);
}
```

図 4.8: 出力制限データ x を総当たりで調べるコード

4.5.3 ロールバック方式

保護観察違反検出方式では，保護観察モード中に非出力制限データを変更した場合，保護観察ビットを立てる．これに対して，ロールバック方式では，以下のような手法をとる．

- 保護観察モード中に非出力制限データを変更する場合，変更前の値を複製，保存する．
- 保護観察モード終了時に，変更された非出力制限データの値を複製しておいた元の値に置き換える．

図 4.6 の例でこの方式を用いた場合，x の値にかかわらず，3 行目の if 文によって移行した保護観察モードが終了した時点では，y は必ず true となる．

ロールバック方式では，保護観察モードの終了時に，保護観察モード中に行った非出力制限データに対する変更が無効化される．そのため，保護観察違反検出方式とは異なり，保護観察モード終了後に，保護観察モード中に発生する暗黙的フローによって情報が漏洩することはない．なおメモリの複製は OS がページ単位で行い，保護観察モード終了時のロールバックも OS が行うことを想定している．

ロールバック方式は，保護観察違反検出方式に比べ，メモリの複製とロールバックによるオーバーヘッドがかかる．この二つのオーバーヘッドは保護観察モード中に変更するメモリページ数に依存する．

マルチスレッドプログラムについて

マルチスレッドプログラムにおいては，通常モードスレッドと保護観察モードスレッドが同時に存在する可能性がある．スレッドはメモリ空間を共有しているため，通常モードスレッドが保護観察モードスレッドの変更した非出力制限データの値を利用できてしまうという問題がある．

この問題に対して，保護観察違反検出方式では，先に述べたように通常モードスレッドが保護観察ビットの立っているデータを利用した時点で保護観察違反とすればよい．一方，ロールバック方式でマルチスレッドプログラムを実行する場合，通常モードスレッドには，複製した (変更前の) メモリページにアクセスさせる必要がある．

4.6 プログラマへのインターフェース

保護観察モードのインターフェースとして，プラグラマに二つのシステムコールを提供する．一つは 4.3 で述べた，保護観察モードの終了アドレスを申告するためのシステムコール (setjoinaddr) であり，二つめは，プログラマが特定の領域のデータに自ら出力制限データとするためのシステムコール (settag) である．

保護観察違反検出方式およびロールバック方式では共に，保護観察モード中に変更した非出力制限データの値を保護観察モード終了後に読むことができない．これは，保護観察モード中のみ使用する自動変数などでは問題にならないが，グローバル変数など，保護観察モード中に値を変更し，保護観察モード終了後もその値を使用したい場合に問題となる．

そこで，プログラマへのインターフェースとして，特定のメモリ領域に出力制限タグを付加するためのシステムコール (settag) を提供する．settag システムコールを用いて，グローバル変数の格納場所などに自ら出力制限タグを付加することで，プログラムはその領域のデータを保護観察モード終了後も使用できるようになる．なお，settag は setjoinaddr と同様，通常モード時にのみ使用可能である．

第5章　実装および評価

x86 エミュレータ Bochs [1] を用いて，提案手法の実装を行った．Bochs はシステムレベルエミュレータであり，プロセッサ，メモリに加えて，グラフィック，ネットワーク，ディスクドライブ等のデバイスをエミュレートする．OS は Red Hat Linux 6.2 を使用し，提案手法を実装した Bochs 上で動作させた．

5.1　タグ管理機構の実装

提案手法では，メモリおよびレジスタを拡張し，出力制限タグを付加する．レジスタに関しては，プロセッサ内部にタグの領域を付加すればよいが，メモリモジュールに関しては，ハードウェア拡張による方法はコストが大きい．そのため，メモリ上のデータに付加するタグは，データ本体とは独立して保存，管理することを考える．

タグは，ページテーブルに似た階層構造のタグテーブルをメモリ上に構築することで管理する．タグの値はアドレス空間においてある程度連続することが多い．そのため，タグテーブルを階層構造にし，配下の領域が全て同じタグであった場合に，それより下位の階層のタグテーブルを省略することで，タグテーブルの容量を圧縮できる．

図 5.1 に，タグテーブルの構成を示す．タグテーブルのエントリーには，次のレベルのテーブルへのポインタ，もしくはタグの値が格納される．タグテーブルは，タグを知りたいデータのアドレスをキーとして参照するが，その際，タグテーブルの階層に対応したアドレスの一部分をインデックスとして使用する．該当エントリにポインタが格納されていた場合には，そのポインタから次の階層のタグテーブルを参照する必要がある．一方，タグが格納されていた場合には，そのタグが求めるタグである．

プロセッサ内部でタグを管理する機構として，Tag Management Unit(TMU) を設計，実装した．図 5.2 に TMU の位置づけおよび構成を示す．TMU は，タグテーブルに対する読み書きを行う Tag Walker と，タグテーブルの一部をキャッシュする Tag Cache から構成される．

プロセッサ内部のキャッシュのうち，L1 キャッシュでは，データとタグをセットで保存し，L2 キャッシュではデータとタグを独立して保存する．Tag Walker は，この L1 キャッシュと L2 キャッシュの間に位置し，L1 キャッシュミス時にタグの解決

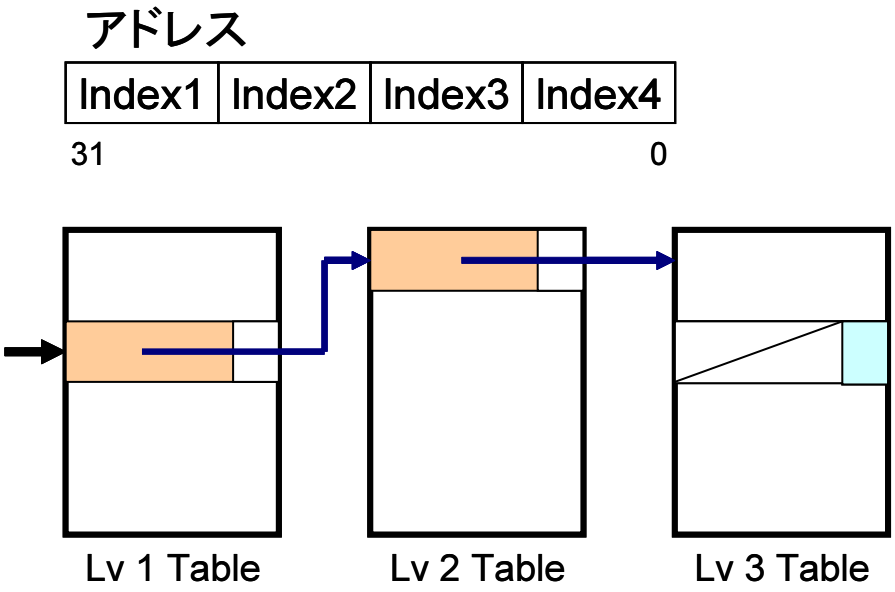

図 5.1: タグテーブル

を行うほか，L1 からのリプレース時にタグの保存を行う．

Tag Walker はメモリ上に展開されたタグテーブルの先頭アドレスから，タグテーブルをたどることで，タグを読み書きする．また，タグの書き込み時には，連続しているタグを検出し，不要なタグテーブルを削除する．

Tag Cache には，タグテーブル参照のキャッシュを格納する．Tag Walker は，アドレスをキーとして Tag Cache を参照し，ヒットすればその値を L1 キャッシュに返す．そのため，タグテーブルは Tag Cache にミスした場合にのみ，たどればよいことになり，タグ参照遅延を抑えることができる．

Tag Cache のエントリーを図 5.3 に示す．なお，V は Valid Bit であり，V が立っている時，そのエントリーは有効である．また，Lv は Tag Cache でキャッシュしているタグテーブルの内容が，何階層目のものであるかを示し，ヒット判定においてマスクとして使用する．Tag Cache のヒットミス判定の様子を図 5.4 を示す．

5.2 情報漏洩防止機能の動作

今回は初期的な実装のため，特定のファイル名のファイルに保護ポリシが設定されていると仮定し，プログラムがそのファイルからデータを読み込んだ場合に出力制限タグが付加される，という形をとっている．以下，いくつかのプログラムを実装システム上で動作させた場合の挙動について説明する．

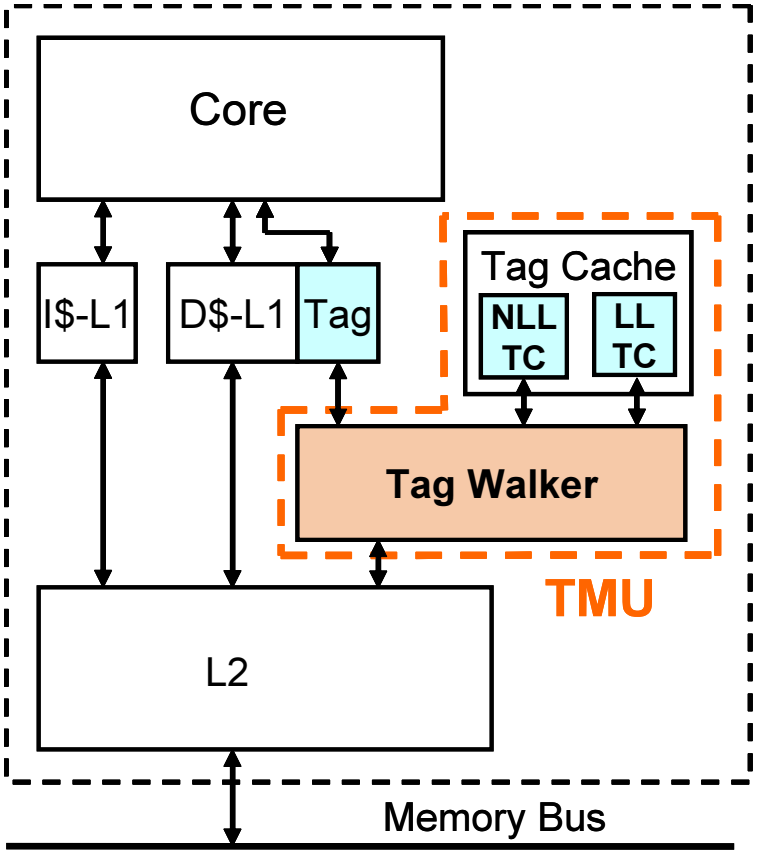

図 5.2: Tag Management Unit

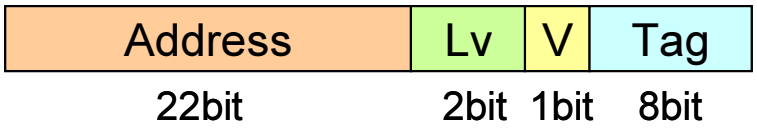

図 5.3: Tag Cache のエントリー

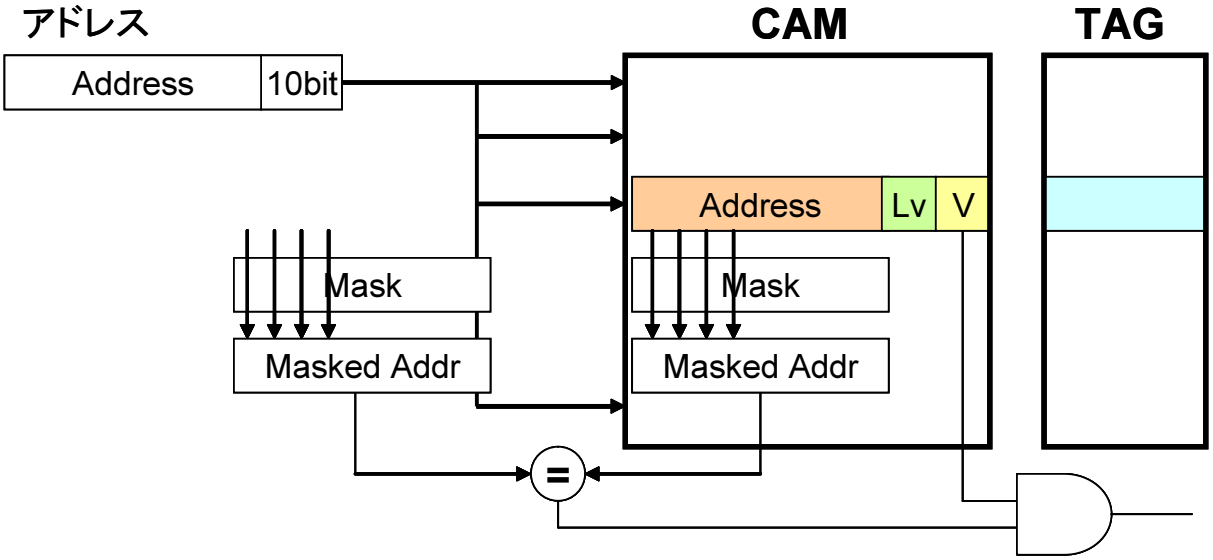

図 5.4: Tag Cache のヒット判定

5.2.1 thttpd

実装システム上で，Web サーバアプリケーション thttpd-2.25b を動作させた．保護ポリシの設定されていないファイル (311byte) をブラウザから表示したときの，実装システムのログは以下のようになる．

WRITE w/o Restriction Tag (fd: 4, at: 0x807dee0, size: 241)

WRITE w/o Restriction Tag (fd: 4, at: 0x807dda0, size: 311)

これは，出力制限タグのついていないデータ (非出力制限データ) の出力を 2 度行ったことを示している．一方，保護ポリシ付きファイル (2402byte) をブラウザから表示しようとしたときの，ログは以下のようになる．

WRITE w/o Restriction Tag (fd: 4, at: 0x807dee0, size: 243)

WRITE with Restriction Tag (fd: 4, at: 0x807e078, size: 2402)

2 回目の出力で，出力制限タグ付きのデータが出力されていることがわかる．これは最も単純な例であるが，保護ポリシ付きのファイルから読み込んだデータが，出力制限データとして確かに追跡されることがわかる．

5.2.2 wc

実装システム上で GNU wc を動作させた．まず，保護ポリシの設定されていないファイルを入力したときのログは以下のようになる．

WRITE w/o Restriction Tag (fd: 1, at: 40019000, size: 34)

これに対して，保護ポリシ付きファイルを入力したときのログは以下のように

なる．

WRITE in Probation Mode (fd: 1, at: 40019000, size 36)

プログラム実行中に保護観察モードへと移行し，保護観察モード中に標準出力への出力を行うことがわかる．保護観察モード中の出力は保護ポリシに従うため，この出力の可否も保護ポリシ依存となる．

5.2.3 jpeg デコーダ

入力された jpeg ファイルをデコードし，bmp ファイルとして出力するプログラム (jpegdec) を作成した．bmp ファイルの出力の可否は，入力された jpeg ファイルの保護ポリシによって決まる．jpegdec は保護観察モードを意識した形で作成しており，setjoinaddr および settag システムコールを使用している．setjoinaddr システムコールは，ヘッダの解析部において移行する保護観察モードを，デコード処理の終了部で終了させるために用いており，settag システムコールは，デコード結果を格納する領域にあらかじめ出力制限タグを付加するために用いている．

今回作成した jpegdec は，デコード終了後，単に bmp ファイルを出力するのみであり，その機能の実現において必ずしも保護観察モードを終了させる必要はない．しかし，デコード処理の終了時に保護観察モードを終了させることで，それ以降システムコールの呼び出し制限を受けなくなり，より複雑な機能を jpegdec に持たせたい場合に，その実現が容易になる．

5.3 オーバーヘッド

提案手法では，出力制限タグや保護観察ビットといったタグをデータに付加する．本節では，タグを付加したことによるオーバーヘッドについて述べる．

5.3.1 メモリオーバーヘッド

出力制限タグのサイズはプログラムが同時に扱うことのできる保護ポリシの数に直結するものであり，今回は 32 ビットのデータに対して保護観察ビットと併せて 8bit のタグを付加している．そのため，単純には 8/32 = 25% のメモリオーバーヘッドが発生するが，5.1 で述べたように，多階層のタグテーブルを用いてタグを管理することで，メモリ消費量を抑えることができる．

二つの異なるサイズの JPEG ファイルを jpegdec でデコードしたときの，データによるメモリ消費量 (data)，単純にタグを保存した場合のタグによるメモリ消費量 (naive)，タグテーブルによるメモリ消費量 (table) を表 5.1 に示す．タグテーブルを用いた場合，単純にタグを保存した場合に比べ，タグの記憶に消費するメモリ量を 90%程度削減できる．

表 5.1: メモリ消費量

	data[KB]	tag(naive)[KB]	tag(table)[KB]	メモリ消費量削減率 [%]
JPEG (5.7KB)	601	150	16	89.4
JPEG (180KB)	2541	635	39	90.7

表 5.2: 評価パラメータ

L1 Data Cache	4KB, 2-way, 64B Line, Write Back
L2 Cache	256KB, 4-way, 64B Line, Write Back
Tag Cache	128 Entry, Full Associative
タグテーブル段数	4 段

5.3.2 性能オーバヘッド

タグを付加したことによる L2 キャッシュへのアクセス回数の増加，および L2 キャッシュのミスによるメモリへのアクセス回数の増加はプログラム実行時の性能オーバーヘッドとなる．

二つの異なるサイズの JPEG ファイルをデコードしたときの L2 アクセス回数を図 5.5 に示す．なお，評価に用いたパラメータは表 5.2 の通りである．グラフはタグを付加しない時 (ベースモデル) のアクセス回数を 1 として正規化している．タグを付加したことによって，L2 キャッシュアクセスは 25%程度増加している．

次に，メモリアクセス回数を図 5.6 に示す．なお，図 5.6 ではタグを付加しない時 (ベースモデル) の回数を 1 として正規化している．L2 キャッシュアクセス回数とは異なり，メモリアクセス回数は 4%程度の増加に抑えられている．

上述のアクセス回数の増加と，L1 キャッシュ，L2 キャッシュのヒット率から，ロード・ストア命令の平均レイテンシを求めると，図**??**のようになる．なお，図 5.6 ではベースモデルの回数を 1 として正規化している．評価に用いた，各メモリ階層のアクセスレイテンシは表 5.3 の通りである．平均レイテンシは，5%程度増加している．

ロールバック手法では，タグの参照によるオーバヘッドに加えて，メモリページの複製および，保護観察モード終了時のロールバックが実行速度のオーバーヘッド要因となる．この二つのオーバーヘッドは，保護観察モード中にいくつのメモリページに渡って非出力制限データの変更を行うかに依存する．

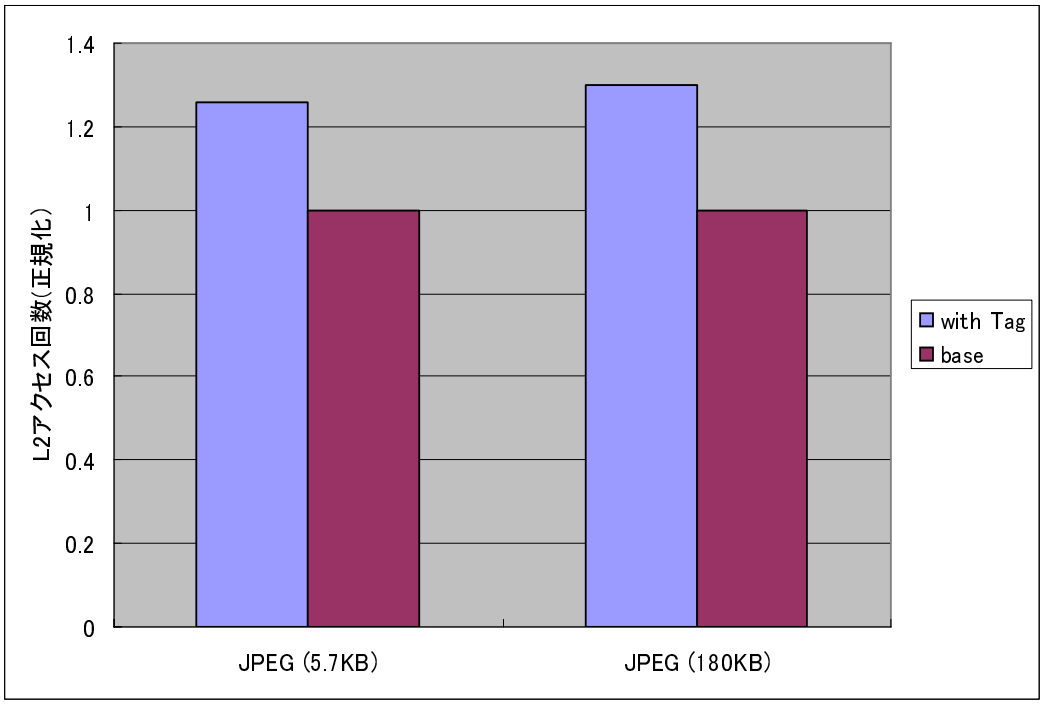

図 5.5: L2 アクセス回数

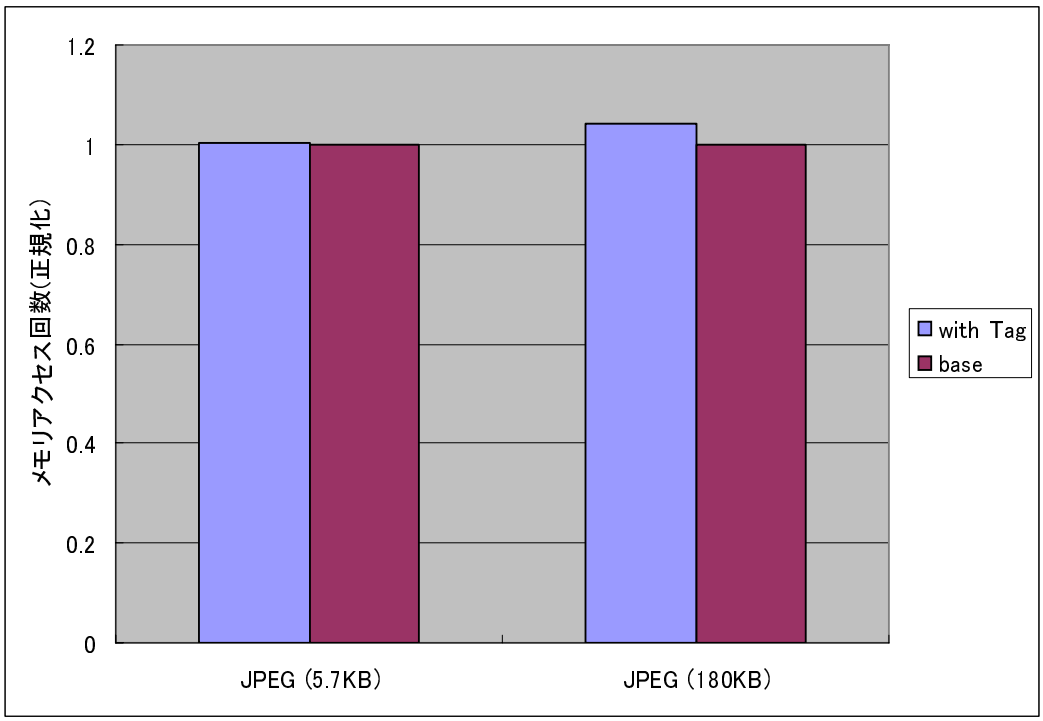

図 5.6: メモリアクセス回数

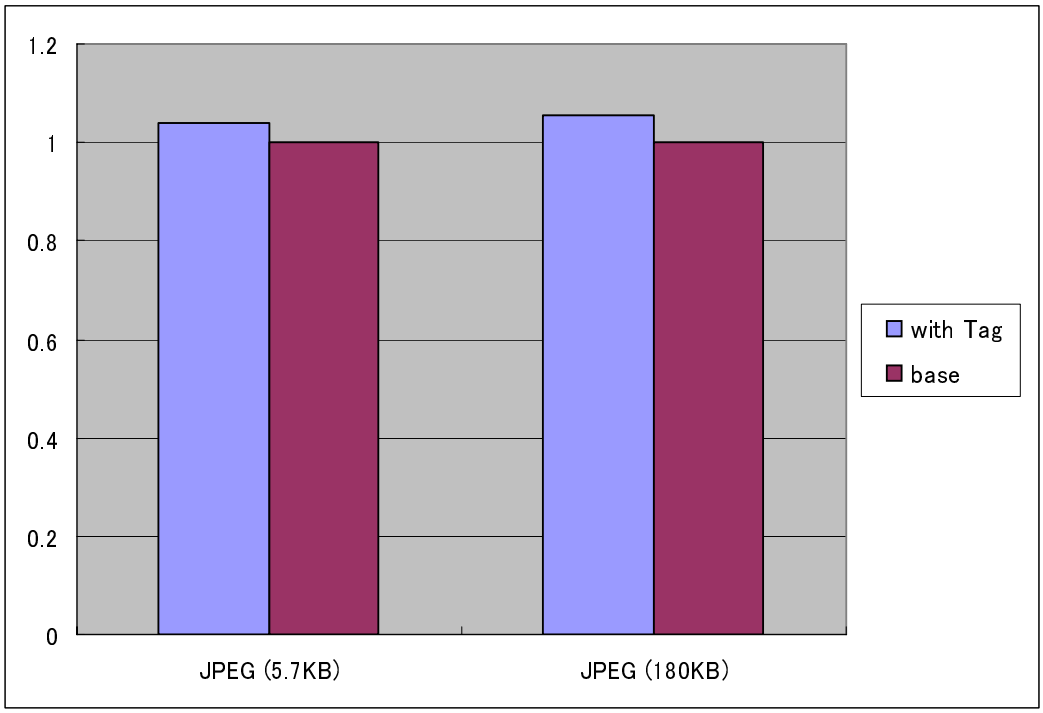

図 5.7: ロード・ストア命令の平均レイテンシ

表 5.3: キャッシュ，およびメインメモリのアクセスレイテンシ

L1 Cache	1 cycle
L2 Cache	10 cycle
Tag Cache	1 cycle
メモリ	150 cycle

第6章　関連研究

本章では，機密情報や商用コンテンツの漏洩防止を目的とした技術，研究について述べる．

6.1　アクセス制御

UNIX 系 OS をはじめとする標準的な OS では，ファイルなど各種リソースへのアクセスに際して，任意アクセス制御 (DAC: Discretionary Access Control) と呼ばれるアクセス制御を行っている．DAC はユーザ ID あるいはグループ ID に基づいてリソースの読み出し，書き込み，実行の各権限を制御する方式であり，アクセス権限の設定はリソースの所有者が行う．

DAC には以下のような問題点がある．

- リソースの所有者がそれぞれ自由にアクセス権限を変更できるため，システム全体の設定を一元的に管理することが困難
- スーパーユーザと呼ばれる特別なユーザ (root) が存在し、root はすべての権限を許可される．そのため root 権限で動作するプログラムが乗っ取られた場合，すべてのリソースが情報漏洩の危機にさらされる．

これに対して，SELinux [16] などのセキュア OS では，強制アクセス制御 (MAC: Mandatory Access Control) と呼ばれるアクセス制御を行う．MAC では，リソースへのアクセス制御に関する設定ファイル (セキュリティポリシー) が用意され，その管理はリソースの所有者ではなく，セキュリティ管理者が一元的に行う．またセキュア OS では，特権ユーザである root が存在しないため，すべてのユーザ，プロセスに対してセキュリティポリシーが適用される．

DAC や MAC などのアクセス制御は，一度リソースへのアクセスを認めてしまった場合，それ以降のプログラムの挙動に対しては一切制御できない．そのため，プログラムが入力されたデータを漏洩するか否かはプログラムの信頼性に依存し，ユーザは一方的にプログラムを信頼するしかない．

履歴を用いたアクセス制御方式として，モバイルコードの安全な実行を目的とした Deeds [5] がある．Deeds は，監視や保護の対象となるリソースへのアクセス要求をセキュリティイベントとして定義し，プログラム毎にセキュリティイベン

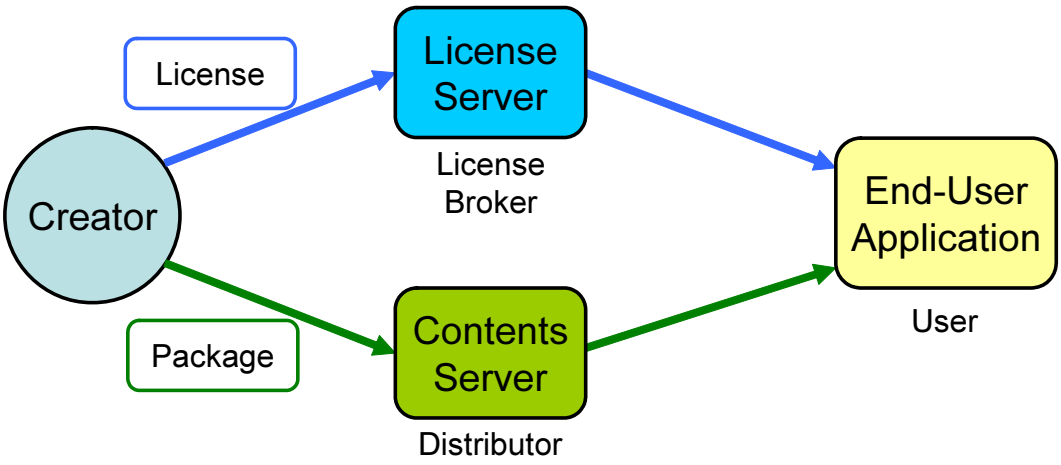

図 6.1: DRM System Flow

トの履歴を保存する．そして新たなアクセス要求がなされた時には，セキュリティイベントの履歴に基づいたアクセス制御を行う．そのため，潜在的に危険なアクセスを行ったプログラムに対しては，以降のアクセスを制限するなどといったことができる．しかし，履歴を用いた保護ポリシを細かく記述することは困難であり，プログラムの挙動を大幅に制限してしまう恐れがある．

6.2 ディジタル著作権管理

ディジタル著作権管理 (Digital Rights Management, DRM) システムでは，ディジタルコンテンツの著作権者がユーザに対して特定の使用規約を履行させるためのメカニズムが実現される [7]．DRM システムはインターネットに接続された PC の使用を前提としたソフトウェアベースのシステムであり，そこではパッケージとライセンスという概念が利用される (図 6.1)．DRM システムの例としては，Windows Server 2003 に実装されている Windows RM(Rights Management) [10] や Apple が開発した iTunes (FairPlay)，Adobe 社の Adobe DRM などがある．本節では DRM システムについて，パッケージとライセンスの視点から説明する．

6.2.1 パッケージおよびライセンスの作成

ディジタルコンテンツはユーザからの不当なアクセスを防止するための保護を行う必要がある．そのため DRM システムでは，作成されたディジタルコンテンツを DES や AES，あるいはシステム独自のアルゴリズムによって暗号化する．また，ディジタルコンテンツの種類によっては，暗号化に先立って電子透かしを埋め込むことがあり，電子透かしによってデータの複製や改変の検出が可能になる．

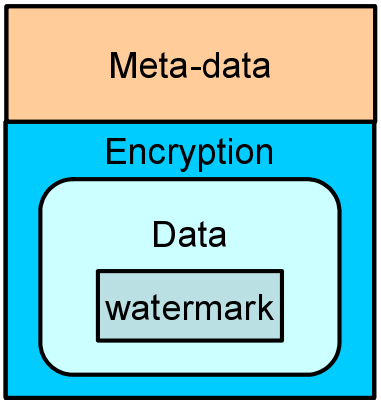

図 6.2: Package

このような保護が施されたディジタルコンテンツに対して，識別用のメタデータとライセンスの入手に関する情報を加えたものをパッケージと呼ぶ (図 6.2)．DRM システムでは，このパッケージが権利管理の最小単位となる．

ディジタルコンテンツの著作権者は，ユーザに与える権利とその条件を明確にする必要がある．そのため，一般に DRM システムでは Rights Expression Language(REL) と呼ばれる権利記述言語が使用される [15]．REL は XrML [4] や ODRL [2] などをはじめとしていくつかの種類があるが，いずれも XML をベースにしている．REL による権利記述は基本的に以下の 4 つの要素から構成される．

Resource 権利記述の対象となるディジタルコンテンツの識別情報.

Parties ユーザや流通事業者などディジタルコンテンツに関わる主体．特定の復号鍵を持っているか否かによって識別される．

Rights Resource に対して Parties が許可される権利．具体的には，データの再生，印刷，複製，改変，ライセンスの変更，再発行などが含まれる．

Conditions Rights で示された権利を行使する際に満たされていなければならない必要条件．具体的には，有効期間，制限回数，課金方式 (定額，ペイパービュー) などが含まれる

XrML による権利記述の例を図 6.3 に示す．

ライセンスはこの REL による権利記述と暗号化された復号鍵，および認証情報から構成される．ユーザがパッケージ化されたディジタルコンテンツを利用する際に，このライセンスが必要となる．ライセンスの不当な複製によるディジタルコンテンツへのアクセスを防ぐため，復号鍵はユーザ毎に異なる鍵によって暗号化が施される．これをライセンスの個別化という．暗号化された復号鍵を復号できるのは，信頼された DRM システムの構成要素であるエンドユーザアプリケー

```
<keyHolder licensePartID="Alice">
  <info>
    <dsig:KeyValue>
      <dsig:RSAKeyValue>
        <dsig:Modulus>oRUTUiTQk… /dsig:Modulus>
        <dsig:Exponent>AQABAA==</dsig:Exponent>
      </dsig:RSAKeyValue>
    </dsig:KeyValue>
  </info>
</keyHolder>
<mx:play/>
<mx:diReference>
  <mx:identifier>
    urn:PDQRecords:song:WhenTheThistleBlooms.mp3
  </mx:identifier>
</mx:diReference>
<validityInterval>
  <notBefore>2003-02-13T15:30:00</notBefore>
  <notAfter>2003-03-13T15:30:00</notAfter>
</validityInterval>
```

図 6.3: An Example of Rights Expression

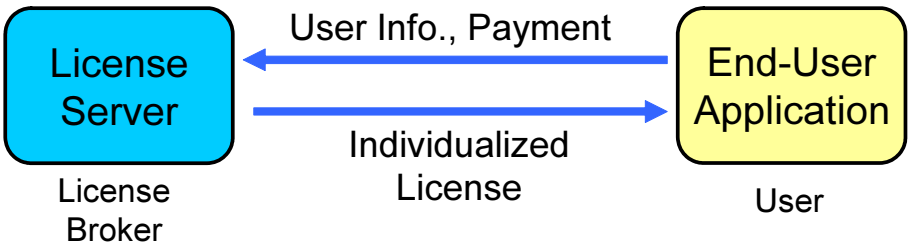

図 6.4: License Server

ションのみである．認証情報には，ライセンスの発行者に関する情報とライセンス全体の完全性を保証するための電子署名が含まれる．

6.2.2　パッケージおよびライセンスの配布

パッケージとライセンスはユーザに対して配布，発行する必要がある．パッケージは暗号化されているため，その配布形態は自由である．それに対してライセンスの配布は，以下の二つの点でパッケージの配布と異なる．

- ライセンスはユーザ毎に個別化される必要がある．
- ライセンスのやりとりには一般に金銭的な取引が伴う．

このため，DRM システムでは一般にライセンスブローカーの管理するライセンスサーバが導入される．ライセンスブローカーはディジタルコンテンツの著作権者とユーザの間に入り，両者のライセンスのやりとりを仲介する．また，必要であればユーザに対する課金も代行する．このライセンスブローカーの働きにより，ディジタルコンテンツの作成者はライセンスの発行に関する負担から解放される．ライセンスサーバは信頼された DRM システムの構成要素として，ライセンスの内容を一部変更して再発行する権利が最初に作成者が発行するライセンスに基づいて与えられる．ライセンスサーバはこの権利を行使することによって，ユーザ毎にライセンスを個別化することが可能になる (図 6.4).

6.2.3　ライセンスの使用

パッケージとライセンスを入手したユーザは，認証されたエンドユーザアプリケーションを利用して，ライセンスに規定された範囲内でディジタルコンテンツにアクセスすることができる．ここでいうエンドユーザアプリケーションは広義には，Windows Media Player のようなソフトウェアに加えて，iPod のようなハードウェアも含まれるが，ここでは基本的にソフトウェアのみを想定する．

これまで，ライセンスはユーザ毎に個別化されると述べてきたが，正確にはユーザが使用するエンドユーザアプリケーションに対して個別化される．そのため，ライセンスの発行に際してエンドユーザアプリケーションはDRMシステムから認証を受け，各ユーザ環境におけるエンドユーザアプリケーションはそれぞれ別個のものとしてシステムから認識される．つまり，ユーザは特定のエンドユーザアプリケーションに対するライセンスをそのまま他の環境に流用することはできない．

既存のDRMシステムはエンドユーザアプリケーションの仕様を公開していないため，DRMシステムの提供者以外がDRMシステムによって認証されるエンドユーザアプリケーションを作成することは不可能である．そのためユーザは，DRMシステムによって指定される独占的なエンドユーザアプリケーションを利用せざるを得ない．

6.3 インフォメーションフロー解析

6.3.1 プログラミング言語ベースの方式

プログラムのインフォメーションフローをコンパイラによって解析する研究がある [11–13]．これらの研究では，専用の型安全なプログラミング言語を提案，提供し，プログラム作成者はその専用言語を用いてプログラムのソースを記述する．Myersによる JFlow [12] はそのような専用言語の一つであり，Javaを拡張したものである．

それらの専用言語では，データおよびデータの格納先に対してラベルが割り当てられる．ラベルはそのデータの保護ポリシ，つまりデータがどのような保護を受けるのかを示すものであり，ラベルの初期化はプログラマが行う．このときインフォメーションフローは，データのラベルとそのデータの格納先のラベルの組み合わせによって表現される．

コンパイラはソースコード中の全てのインフォメーションフロー (明示的フローおよび暗黙的フロー) を静的に解析し，プログラマの決めた保護ポリシに違反するインフォメーションフローを検出した場合に，それを提示するこれにより，保護ポリシに違反しないプログラムの開発が可能となる．

こうしたプログラミング言語ベースのインフォメーションフロー解析技術における最大の問題点は，保護ポリシの決定権がプログラムの開発者にあることである．コンパイラはあくまでプログラム開発者の意図しないインフォメーションフローが存在しないことを検証するのみであり，プログラムを実際に使用するユーザやコンテンツの著作権者の意図は反映されない．そのため，プログラムが実際に情報を漏洩しないか否かは結局プログラム作成者の信頼性に依拠することになる．また当然，専用言語で開発されたプログラムでしかデータの保護は保証されない

Base ISA Instruction	Base ISA semantics	IFS ISA Instruction	Augmented ISA semantics
regop R[a]=R[b],R[c]	R[a] := R[b] *op* R[c]	<S[j],...>regop R[a]=R[b],R[c]	$\underline{R[a]}$:= $\underline{R[b]}$ ⊕ $\underline{R[c]}$ ⊕ S[j] ⊕ ...
load R[a]=[R[b]]	R[a] := Mem[R[b]]	<S[j],...>load R[a]=[R[b]]	$\underline{R[a]}$:= $\underline{Mem[R[b]]}$ ⊕ $\underline{R[b]}$ ⊕ S[j] ⊕ ...
store [R[a]]=R[b]	Mem[R[a]] := R[b]	<S[j],...>store [R[a]]=R[b]	$\underline{Mem[R[a]]}$:= $\underline{R[a]}$ ⊕ $\underline{R[b]}$ ⊕ S[j] ⊕ ...
(R[a])branch T	if(R[a]) jump to T	(R[a])branch T	-
-	-	<S[j],...>join S[a]=S[b],S[c]	S[a] := S[b] ⊕ S[c] ⊕ S[j] ⊕ ...

図 6.5: RIFLE のバイナリ変換

6.3.2 バイナリ変換による方式

RIFLE

Vachharajani らによる RIFLE [18] は，バイナリ変換と専用プロセッサを組み合わせた技術である．バイナリ変換では，インフォメーションフローを追跡するためのコードを追加し，変換後のバイナリを専用プロセッサで実行することで，動的にインフォメーションフローを追跡する．

RIFLE のバイナリ変換では，通常の命令セットアーキテクチャ(ISA) から，インフォメーションフローセキュリティ ISA(IFS ISA) への変換が行われる．具体的には，暗黙的フローを明示的フローに変換するコードと明示的フローによるセキュリティ情報を伝搬するコードが追加される．実際の変換ルールを図 6.5 に示す．ただし，R[i] は汎用レジスタ，S[i] は専用プロセッサが持つセキュリティレジスタをそれぞれ表し，$\underline{x}$ はデータ x のラベルを示す．

このバイナリ変換によってインフォメーションフローはすべて明示的なものとなり，専用プロセッサで実行した場合，自動的にインフォメーションフローが追跡される．

プログラミング言語ベースの技術がソースコードに対して静的な解析を行う一方で，RIFLE は実行バイナリを静的に解析する．そのため，保護ポリシの決定においてプログラム開発者の影響は排除され，プログラムに依存しないユーザ指向の情報漏洩防止が実現可能である．

RIFLE では暗黙的フローを明示的フローに変換しているが，これは，条件分岐命令に依存する命令を静的に解析し，それらの命令に明示的にタグ付けを行うことで実現している．しかし，そうした解析にはメモリ依存解析が必要となり，厳密に行うことはできない．そのため，プログラムのデータ出力が過剰に制限され，プログラムが本来もつ機能が失われる恐れがある．

LIFT

Qin らによる LIFT [14] は，バイナリ変換によって，ソフトウェアでインフォメーションフローを追跡するアプローチをとっている．そのため，LIFT は特別なハードウェアを必要としない．

LIFT は，情報漏洩の防止にとどまらず，より幅広いソフトウェアの脆弱性の克

服を目的としている．ソフトウェアの脆弱性としては，バッファーオーバーフローやフォーマットストリングなどの脆弱性が知られているが，LIFT はこれらの脆弱性に対する攻撃を，安全でないデータによるコントロールフローの変化として検出する．

しかし，情報漏洩の防止を考えた場合，LIFT の機能は十分ではない．それは，LIFT が対象としているインフォメーションフローが明示的フローのみであり，暗黙的フローによる情報漏洩には全く対応できないためである．

第7章　おわりに

7.1　本論文のまとめ

本論文では，プログラム実行時のインフォメーションフローを動的に追跡，制御するプラットフォームによって，情報漏洩の防止が可能であることを示し，具体的なインフォメーションフローの制御手法を提案した．

プログラム実行時の情報漏洩は，プログラムに入力された機密情報が不正に外部に出力されたときに発生する．そのため，プログラム実行環境(プラットフォーム)がプログラム実行時の情報の流れであるインフォメーションフローを追跡・制御し，プログラムが出力しようとするデータとプログラムに入力されたデータとの依存関係がわかれば，出力の正当性を検証し，情報漏洩を防止することができる．

プログラム実行時のインフォメーションフローは大きく，データフローに付随する明示的フローとコントロールフローに由来する暗黙的フローの二つがある．このうち明示的フローに関しては，命令の実行時にタグを伝搬させることによって比較的容易に追跡が可能である．

一方，暗黙的フローは，命令を実行しないことによっても発生するため，その動的な追跡はきわめて難しい．そこで本研究では，暗黙的フローが特定の分岐からその合流までの間に発生することに着目し，この区間を保護観察モードとして通常の実行モードと区別した．保護観察モード中はシステムコールの呼び出しを制限し，保護観察モード終了後には保護観察モード中に変更したメモリ領域へのアクセスを禁止することによって，暗黙的フローを介した情報漏洩を防止している．

提案手法は，プログラムの信頼性に依存しない情報漏洩の防止を可能にするものであり，プログラマへのインターフェースの提供によって，プログラム機能の柔軟性も維持する．

システムレベルエミュレータ Bochs を用いて提案手法の初期的な実装を行い，その機能を評価した．提案手法では，タグの追加によって，L2 キャッシュおよびメモリへのアクセス回数が増加する．L2 キャッシュへのアクセス回数はタグがない場合の 2.5 倍程度であったが，メモリへのアクセス回数は 2%程度の増加であった．また，提案プラットフォーム上で実行されることを想定したプログラムとして JPEG デコーダを開発し，実用的なプログラムが提案プラットフォーム上で動作可能であることを示した．

7.2 今後の課題

今後の課題については以下のものが挙げられる．

- インターフェースの整備

  本論文では，インフォメーションフローの追跡・制御に必要となるプロセッサの機能を中心に検討を行ったが，プラットフォームとして機能させるためには，さらなる OS の機能追加，および変更が必要となる．特に，保護ポリシの解釈は情報漏洩防止機能の実現において重要である．また，提案プラットフォーム上で実行するプログラムを開発する際には，保護観察モード特有のメモリアクセスに関する制限を意識しなければならないため，プログラマに対する補助ツールの提供も求められる．

- 正確なオーバーヘッドの評価

  本論文では，主にタグの参照に関するオーバーヘッドについてのみ評価を行った．しかし，提案手法ではシステムコールの検証によるオーバーヘッドがあるほか，ロールバック手法に関しては，メモリの複製，ロールバックによるオーバーヘッドもある．これらのオーバーヘッドについては，OS の機能を実装し，定量的な評価を行うことが求められる．

- 静的な手法との融合

  本論文では，インフォメーションフローを動的に追跡・制御する手法を提案した．その最大の理由は，プログラムの信頼性を前提とせずに情報漏洩を防止するためであるが，もう一つの理由として，プログラムバイナリに対する静的なインフォメーションフロー解析が正確に行えないことが挙げられる．しかし，不十分な解析であったとしても，バイナリに対する静的な解析の結果を，保護観察モードの自動的な終了などに利用できると考えられる．

参考文献

[1] Bochs: the Open Source IA-32 Emulation Project. http://bochs.sourceforge.net.

[2] Open Digital Rights Language(ODRL). http://www.odrl.net/.

[3] Trusted Computing Group(TCG). https://www.trustedcomputinggroup.org/.

[4] ContentsGuard. eXtensible rights Markup Language(XrML) 2.0 Specification. http://www.xrml.org/.

[5] G. Edjlali, A. Acharya, and V. Chaudhary. History-based access control for mobile code. In *CCS '98: Proceedings of the 5th ACM conference on Computer and communications security*, pages 38–48, New York, NY, USA, 1998.

[6] T. Hauser and C. Wenz. Drm under attack: Weaknesses in existing systems. In *Digital Rights Management*, pages 206–223, 2003.

[7] P. A. Jamkhedkar and G. L. Heileman. Drm as a layered system. In *DRM '04: Proceedings of the 4th ACM workshop on Digital rights management*, pages 11–21, New York, NY, USA, 2004. ACM Press.

[8] W. Ku and C.-H. Chi. Survey on the technological aspects of digital rights management. In *ISC*, pages 391–403, 2004.

[9] Q. Liu, R. Safavi-Naini, and N. P. Sheppard. Digital rights management for content distribution. In *CRPITS '03: Proceedings of the Australasian information security workshop conference on ACSW frontiers 2003*, pages 49–58, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc.

[10] Microsoft. Technical Overview of Windows Rights Management Services. http://www.microsoft.com/windowsserver2003/ techinfo/overview/rmenterprisewp.mspx.

[11] A. Myers and A. Sabelfeld. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[12] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. 26th ACM Symposium on Principles of Programming Languages*, 1999.

[13] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Symposium on Operating Systems Principles*, pages 129–142, 1997.

[14] F. Qin, C. Wang, Z. Li, H. seop Kim, Y. Zhou, and Y. Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *Proceedings of the 39th International Symposium on Microarchitecture (MICRO)*, pages 135–148, 2006.

[15] Y. Qu, X. Zhang, and H. Li. Orel: an ontology-based rights expression language. In *WWW Alt. '04: Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 324–325, New York, NY, USA, 2004. ACM Press.

[16] S. Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. *Proceedings of the FREENIX Track: USENIX Annual Technical Conference*, 2001.

[17] M. L. Smith. Digital rights management & protecting the digital media value chain. In *MUM '04: Proceedings of the 3rd international conference on Mobile and ubiquitous multimedia*, pages 187–191, New York, NY, USA, 2004. ACM Press.

[18] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. Rifle: An architectural framework for user-centric information-flow security. In *Proceedings of the 37th International Symposium on Microarchitecture (MICRO)*, December 2004.

発表文献

主著論文

- 動的なインフォメーションフロー制御による情報漏洩防止手法
  栗田 弘之, 入江 英嗣, 五島 正裕, 坂井 修一
  情報処理学会 研究会報告 ARC 2007-ARC-172(Mar. 2007)(発表予定)

- 動的なインフォメーションフロー制御による情報漏洩防止手法
  栗田 弘之, 入江 英嗣, 五島 正裕, 坂井 修一
  先進的計算基盤システムシンポジウム SACSIS 2007, (May. 2007)(投稿中)

共著論文

- 超ディペンダブル・プロセッサアーキテクチャの構想
  入江 英嗣, 荻野 健, 勝沼 聡, 清水 一人, 栗田 弘之, 五島 正裕, 坂井 修一
  電子情報通信学会技術研究報告 CPSY2006-1〜12, Vol.106, No.3, pp.49-54 (Apr. 2006)

- アドレスオフセットに着目したデータフロー追跡による注入攻撃の検出
  勝沼 聡, 栗田 弘之, 塩谷亮太, 清水 一人, 入江 英嗣, 五島 正裕, 坂井 修一
  先進的計算基盤システムシンポジウム 2006(SACSIS2006), Vol.2006, No.5, pp.515-524 (May. 2006)

- Base Address Recognition with Data Flow Tracking for Injection Attack Detection
  Satoshi Katsunuma, Hiroyuki Kurita, Ryota Shioya, Kazuto Shimizu, Hidetsugu Irie, Masahiro Goshima and Shuichi Sakai
  IEEE International Symposium on Pacific Rim Dependable Computing (PRDC 2006), pp.165-172 (Dec. 2006)

謝辞

非常に多くの方々から多大なご指導、ご協力、励ましを頂き、本論文を完成させることができました。この場を借りて、感謝の意を表したいと思います。

本研究を進めるにあたり，指導教官である坂井修一教授には，2 年間に渡って多くのご指導，ご助言を頂きました.

また，五島正裕助教授からも，大変多くのご指導を頂きました．ここに深く感謝の意を表します.

入江英嗣博士には研究チームのリーダーとして，様々な形でアドバイスを頂きました．清水一人氏，勝沼聡氏，塩谷亮太氏をはじめ，ディペンダブル・グループのメンバーの皆様には，ミーティングにおける議論を通して，貴重なご意見を頂きました.

八木原晴水さん，月村美和さんには，研究室における設備の導入や各種事務手続きなど，研究室で過ごすための様々なご支援を頂きました.

本研究は，21 世紀 COE「情報科学技術戦略コア」，及び，科学技術振興機構 CREST,「ディペンダブル情報基盤」による支援を受けて行われました.