NAIST-IS-DD0561035

博士論文

秘密隠蔽のための系統的ソフトウェア保護フレームワーク

山内 寛己

2010年2月4日

奈良先端科学技術大学院大学
情報科学研究科 情報システム学専攻

本論文は奈良先端科学技術大学院大学情報科学研究科に
博士 (工学) 授与の要件として提出した博士論文である.

山内 寛己

審査委員：

松本 健一 教授　（主指導教員）
関 浩之 教授　（副指導教員）
門田 暁人 准教授　（副指導教員）

秘密隠蔽のための系統的ソフトウェア保護フレームワーク*

山内 寛己

内容梗概

近年，ソフトウェアに含まれる秘密の漏洩を防止することの必要性が増大しており，エンドユーザによるソフトウェアシステムに対する攻撃を妨げることが急務となっている．従来，多種様々なソフトウェア保護技術が提案されてきたが，これら多くの技術をどのように使い分け，もしくは併用すべきかについての系統的な方法は，ほとんど議論がされていない．

本論文では，ソフトウェアシステムの系統的な保護を目的として，まず，エンドユーザによるソフトウェアシステムに対する攻撃方法とその対策について整理し，その結果に基づいて，ソフトウェアの各開発工程において段階的にプロテクション技術を適用するためのガイドライン（段階的プロテクション）を提案する．

次に，段階的プロテクションの実施手順において重要となるソフトウェア難読化に着目し，ソフトウェア難読化手法を適材適所に適用するための枠組み（難読化フレームワーク）を提案する．提案フレームワークでは，攻撃者の攻撃におけるゴールを定義するとともに，ゴール達成に必要なサブゴールを，ゴール分解により求めていくことでゴール木を生成する．そして，得られたゴール木の全ての末端のサブゴールについて，その達成を妨げるのに必要な難読化手法を選定する．ケーススタディとして，秘密を含む典型的な Digital Rights Management (DRM) ソフトウェアの 1 つである cryptomeria cipher (C2) 暗号プログラムにおいて，復

* 奈良先端科学技術大学院大学 情報科学研究科 情報システム学専攻 博士論文, NAIST-IS-DD0561035, 2010 年 2 月 4 日.

号鍵を隠蔽するためのゴール木を生成する事例を通して，多数の難読化手法を適材適所に適用できることを示した．

さらに上記の難読化フレームワークにおいて，ゴール木の各ノードを構成する攻撃者の行動に着目すると，全ての行動は，プログラム中から秘密情報，もしくは秘密情報の発見の手がかりとなる情報を探す，といった行動であり，ゴール木は事実上手がかりの連鎖を表す木となっている．そこで，プログラム中の攻撃の手がかりを網羅的に列挙し，それらを難読化によって隠蔽するため，フレームワークの拡張を行う．これにより，秘密情報とその手がかりとの関係，及び，手がかり間の関係を，アルゴリズム，ソースコード，バイナリの 3 つの抽象レベルに分けて記述し，各レベルにおいて難読化により手がかりを隠蔽することで，秘密情報の発見を困難にすることができる．

キーワード

ソフトウェア難読化, プログラム解析, 情報隠蔽, ゴール指向分析, ゴール分解

A Framework for Systematic Software Protection for Secret Hiding*

Hiroki Yamauchi

Abstract

The protection of software has become vital to today's computer systems from the perspective of hiding secrets involved in software. To hide secrets included in software, various types of techniques have been proposed. However, there is no clear guideline for proper use of these techniques.

This paper firstly gives a survey of end-user attacks and protection techniqes, and proposes a guideline on applying proper protection techniques step by step in each software development phase.

Next, this paper focuses on software obfuscation tehniques, which are intended to directly hide secrets in software. For a systematic use of obfuscations, this paper proposes a goal oriented approach to obfuscation. Specifically, we identify the cracker's goal, conduct a goal-oriented analysis, select obfuscations to disrupt all subgoals, and apply selected obfuscations to software. As a case study, we defined a security goal for a cryptomeria ciper (C2) program, typically used in Digital Rights Management (DRM) systems, and demonstrated how the goal oriented analysis was conducted and obfuscation techniques were applied to places where they are needed in hiding decryption keys.

Then we extend our framework to make it easy to conduct the goal oriented analysis, by providing a notation to describe the relationship between clues to

* Doctoral Dissertation, Department of Information Systems, Graduate School of Information Science, Nara Institute of Science and Technology, NAIST-IS-DD0561035, February 4, 2010.

achieve subgoals. In the extended framework, relationships between secrets and their clues are described in three abstraction levels (algorithm, source code and binary code) so that obfuscation techniques can be comprehensively applied in each abstraction level.

Keywords:

software obfuscation, program analysis, information hiding, goal-oriented analysis, goal decomposition

関連発表論文

学術論文誌

1. Hiroki Yamauchi, Akito Monden, Masahide Nakamura, Haruaki Tamada, Yuichiro Kanzaki and Ken-ichi Matsumoto, "A Goal-Oriented Approach to Software Obfuscation," *International Journal of Computer Science and Network Security*, Vol. 8, No. 9, pp. 59–71, September 2008. (第 5 章に関連する)

国際会議発表

1. Hiroki Yamauchi, Yuichiro Kanzaki, Akito Monden, Masahide Nakamura and Ken-ichi Matsumoto, "Software Obfuscation from Crackers' Viewpoint," In *Proceedings of IASTED International Conference on Advances in Computer Science and Technology (IASTED ACST 2006)*, pp.286–291, January 2006. Puerto Vallarta, Mexico. (第 4, 5 章に関連する)

査読付き国内研究集会発表

1. 山内 寛己, 神崎 雄一郎, 門田 暁人, 中村 匡秀, 松本 健一, "マルチバージョン生成によるプログラムの解析防止," 野呂昌満, 山本晋一郎 (編), ソフトウェア工学の基礎 XI, 日本ソフトウェア科学会 FOSE2004, pp. 157–160, November 2004. (第 3 章に関連する)

国内研究集会発表

1. 山内 寛己, 神崎 雄一郎, 門田 暁人, 中村 匡秀, 松本 健一, "攻撃タスクを考慮した難読化による暗号プログラムの保護," 電子情報通信学会技術報告, ソフトウェアサイエンス研究会, No. SS2005-12, pp. 25–30, December 2005. (第 4, 5 章に関連する)

テクニカルレポート

1. 山内 寛己, 門田 暁人, 松本 健一, “実行系列差分攻撃によるプログラムの耐タンパー性評価,” Information Science Technical Report NAIST-TR200907 ISSN 0919-9527, Graduate School of Information Science, Nara Institute of Science and Technology, December 2009. (第 2 章に関連する)

その他の発表論文

国内研究集会発表

1. 石井 健一, 串戸 洋平, 山内 寛己, 井垣 宏, 玉田 春昭, 中村 匡秀, 松本 健一, “異なる設計・実装法を用いたWebサービスアプリケーションの開発および比較評価,” 電子情報通信学会技術研究報告, ネットワークシステム研究会, No. NS2003-315, pp. 107–112, March 2004.

2. 串戸 洋平, 石井 健一, 山内 寛己, 井垣 宏, 玉田 春昭, 中村 匡秀, 松本 健一, “Webサービスアプリケーションのソフトウェアメトリクスに関する考察,” 電子情報通信学会技術研究報告, ネットワークシステム研究会, No. NS2003-316, pp. 113–118, March 2004.

3. 西岡 隆司, 山内 寛己, 門田 暁人, 中村 匡秀, 松本 健一, “類似した命令列の畳込みによるプログラムの耐タンパ性の向上,” 情報処理学会研究報告, ソフトウェア工学研究会, Vol. 2007-SE-155, No. 33, pp. 167–174, March 2007.

4. 岡原 聖, 真鍋 雄貴, 山内 寛己, 門田 暁人, 松本 健一. 井上 克郎, “コードクローンの長さに基づくプログラム盗用確率の実験的算出,” 電子情報通信学会技術報告, ソフトウェアサイエンス研究会, No. SS2008-10, pp. 7–11, October 2008.

5. 岡原 聖, 真鍋 雄貴, 山内 寛己, 門田 暁人, 松本 健一, “ソースコード流用のコードクローンメトリクスに基づく検出手法,” 電子情報通信学会技術報告, 知能ソフトウェア工学研究会, No. KBSE2009-11, pp. 73–78, November 2009.

目 次

図 目 次

表 目 次

第1章　はじめに

近年，様々な秘密がソフトウェア製品に含まれるようになっており，ソフトウェア解析による秘密漏洩を防止することの重要性が高まっている．隠蔽すべき秘密情報としては，次のものが挙げられる．

- システムの安全性に関わる，もしくは，商業的に価値のあるアルゴリズムやサブルーチン．例えば， DVD-Video のデジタルコピー防止用の暗号化規格 CSS (Content Scrambling System) の暗号アルゴリズムは本来非公開であり，システムのユーザに知られないように隠蔽することが要求されていた．

- システムの安全性に関わるデータ（定数）．例えば，携帯電話に含まれる端末識別番号や DRM システムの復号鍵である．CSS の次の世代のデジタルコンテンツ保護規格である CPPM（Content Protection for Prerecorded Media）／CPRM（Content Protection for Recordable Media）では，暗号アルゴリズム（C2 暗号）は公開されているが，コンテンツ再生プレイヤーをソフトウェアで実装する場合に，ソフトウェアに含まれるデバイス鍵（復号鍵の一種），及び，S-Box テーブル（復号に必要な定数の集合）の隠蔽が必須となっている．

- システム内部の機能分岐点．例えば，ライセンスチェックを行う分岐文である．ソフトウェア自身のコピーを防ぐための，いわゆるコピープロテクト技術として，フロッピーディスクや CD-ROM メディアの特殊フォーマットを利用したプロテクトやシリアルナンバープロテクトが知られているが，いずれのプロテクト技術においても「正規のコピーか否かをチェックする」分岐文がソフトウェア内部に存在し，この分岐文が発見，改ざんされると，

プロテクトが無効化される恐れがある．

- 内部インタフェース．例えば，OS が提供するセキュア API へのアクセスなどである．

- 外部インタフェース．例えば，システムへの完全なアクセス権を与えるメンテナンス用インタフェースである．近年では，Borland Interbase SQL データベースサーバのメンテナンス用バックドアが露見した事件が知られている．

これらの秘密情報を隠ぺいするために，数々のソフトウェア難読化技術が提案されてきた．例えば，名前（識別子）難読化，ループ難読化，データ難読化，クラス構造難読化などである [20]．これらの難読化手法は，与えられたソフトウェアを，その仕様を変えずに，より複雑で解析しにくいソフトウェアと変換することによって，解析者 (攻撃者) から秘密を守ることを目的としている．

しかし，それらの難読化手法を個々のソフトウェアに系統的に適用するための枠組みについてはほとんど研究されていない．そのために，どの難読化手法をソフトウェア中のどの部分に用いるべきか不明確であった．また，難読化することによって期待通りの効果が得られるかどうかが不明瞭であった．これらの問題は，多くの難読化手法が目的や攻撃者のターゲットを十分に考慮していないことによる．

また，難読化は，ソフトウェア中の秘密を完全に保護できるほどには強力ではないため，他のソフトウェアプロテクション技術（解析ツール実行を妨げる仕組みの導入，改ざんの検出など）の適用も不可欠である．それらの技術と難読化を併用する枠組みについても十分に研究されていない．

ソフトウェアプロテクションに関するサーベイは，これまでにいくつか存在する [14][18][28][32][43][51][53]．その中でも，van Oorschot のサーベイが詳しい [53]．文献 [53] には，ソフトウェア難読化，Software Diversity，White-box Cryptography，ハードウェアを併用した保護などがサーベイされている．ただし，これらソフトウェアプロテクションの要素技術について網羅性に乏しく，系統的な保護についても述べられていない．また，攻撃のモデルおよび，モデルに応じた攻撃手法などについてもあまり述べられていない．

以上の問題背景から，本論文では，特にソフトウェア単体で実施できる保護手法を対象とし，系統的な適用方法の開発を目指す．まず，これまでに知られているソフトウェア攻撃方法を整理し（第 2 章），それらを防ぐソフトウェアプロテクションの要素技術を整理する（第 3 章）．次に，ソフトウェア開発者の視点から，ソフトウェア開発の各工程において段階的にプロテクション技術を適用することで，外部仕様，内部仕様，アルゴリズム，実装といった，ソフトウェアの各抽象化レベルの弱点を防御する枠組み（段階的プロテクションと呼ぶ）を提案する（第 4 章）．さらに，段階的プロテクションの手順の中で，多様な難読化手法を適材適所に適用するための枠組み（難読化フレームワーク）を提案する（第 5 章）．第 6 章では，難読化フレームワークを拡張し，より網羅的に攻撃方法とその防御方法を整理できる方法を提案する．第 7 章はまとめである．

第2章　ソフトウェアに対する攻撃

ソフトウェアを攻撃から守るためには，まず，どのような攻撃があるのかを知る必要がある．本章では，ソフトウェアに対する攻撃手段を整理する．

1．攻撃のモデル

ソフトウェアシステムに対する攻撃は，図2.1から2.3のような3つに分類できる．図2.1の攻撃モデル1では，攻撃者はソフトウェア開発者であり，エンドユーザの計算機に悪意のあるソフトウェア（マルウェア）をダウンロードさせる．エンドユーザは，無意識にダウンロードしたマルウェアを実行し，マルウェア自身，または，マルウェアがエンドユーザの計算機内の脆弱性を利用し，Code Injection 攻撃するなどによって，計算機内の秘密情報にアクセス可能にする．この場合の被害者はエンドユーザであり，エンドユーザの計算機を守るための対策が必要とされる．代表的な対策としては，Sandbox によって信頼されないプログラムを計算機資源から隔離したり，Code Injection 攻撃を防ぐためにバッファオーバーフロー検出を行うなどが考えられる．

図2.2の攻撃モデル2では，モデル1と異なり，攻撃者はエンドユーザである．エンドユーザは，ネットワーク上でサービスを提供しているサーバ計算機上のソフトウェアに対し，不正な入力を与えることで，サーバ計算機上の秘密情報を得たり，秘密情報の改ざんを行う．この場合の被害者は，サーバ側のソフトウェアの開発者，もしくは，サーバ提供者であり，不正な要求から秘密情報を守るための対策が必要とされる．代表的な対策としては，サーバソフトウェアに対するアクセス制御を行い，その制御方法に問題がないことを検証することや，情報流解析の技術を用いて，秘密情報がエンドユーザに漏洩しないことを検証することな

どが考えられる．

図 2.3 の攻撃モデル 3 は，本論文が対象とする攻撃である．攻撃者であるエンドユーザの計算機上に攻撃対象のソフトウェアがあるため，モデル 1 やモデル 2 のようなネットワーク経由の攻撃と比べて，防御は容易でない．例えば，Sandbox や情報流解析の技術は防御には役立たない．従来，モデル 1 やモデル 2 の攻撃に対しては，攻撃を防ぐための研究分野（ネットワークセキュリティ）が発達しており，攻撃方法や防御方法の解説書が数多く出版されているが，モデル 3 の攻撃の対策は，系統的な対策がほとんど提案されていない．代表的な対策としては，ソフトウェア暗号化，難読化，デバッガ検出などが知られているものの，いずれも攻撃の一側面を妨げるにすぎない．そのため，これら多様なプロテクション技術を適材適所に用いるための枠組みが求められている．

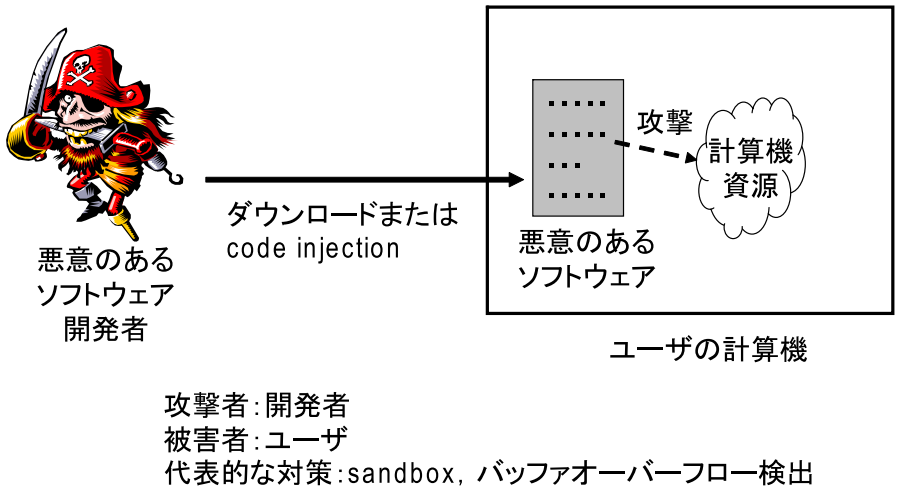


図 2.1 攻撃モデル 1

攻撃モデル 3 において，具体的な攻撃方法は多数考えられる．本章の以降では，それらの攻撃を，解析攻撃，改ざん攻撃など，いくつかに分類して述べる．

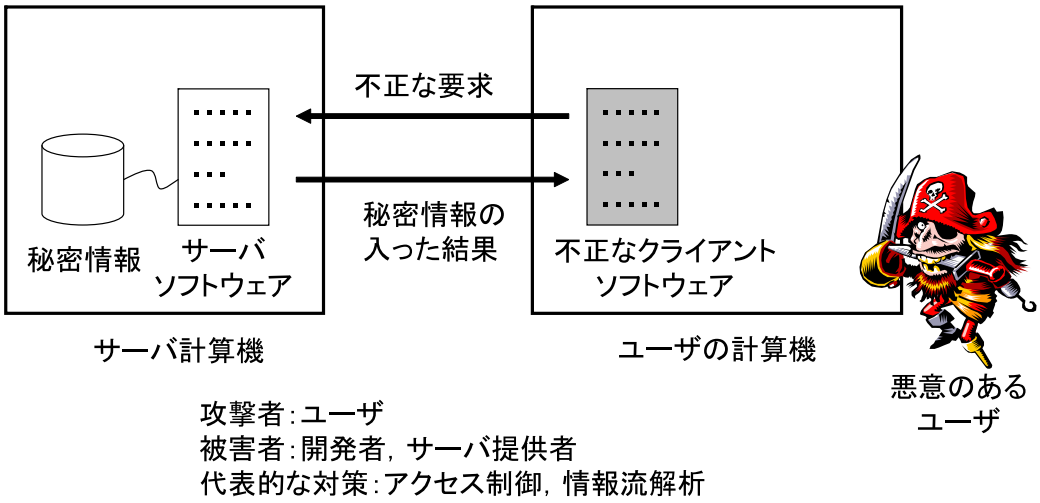


図 2.2 攻撃モデル 2

2. ソフトウェア解析攻撃

一般に，計算機システムおよび，その内部のソフトウェアに対する攻撃は，ブラックボックス攻撃とホワイトボックス攻撃に大別され，さらに，その中間的な攻撃としてサイドチャネル攻撃が知られている [59]．下記では各攻撃とその対策を述べる．

2.1 ブラックボックス攻撃

ブラックボックス攻撃はプログラム内部を解析せずに攻撃する方法であり，プログラムやサブルーチンに対して入力を与え，その出力を観測することでプログラム中の秘密（ライセンスチェックルーチンの動作原理など）を推測する．この攻撃の一種として，リプレイ攻撃 [61]（セキュリティに関する API やライセンスチェックルーチンなどへの入出力を記録し，模倣することでライセンスチェックなどを回避する攻撃）も知られている．攻撃対策として，プログラムを構成する

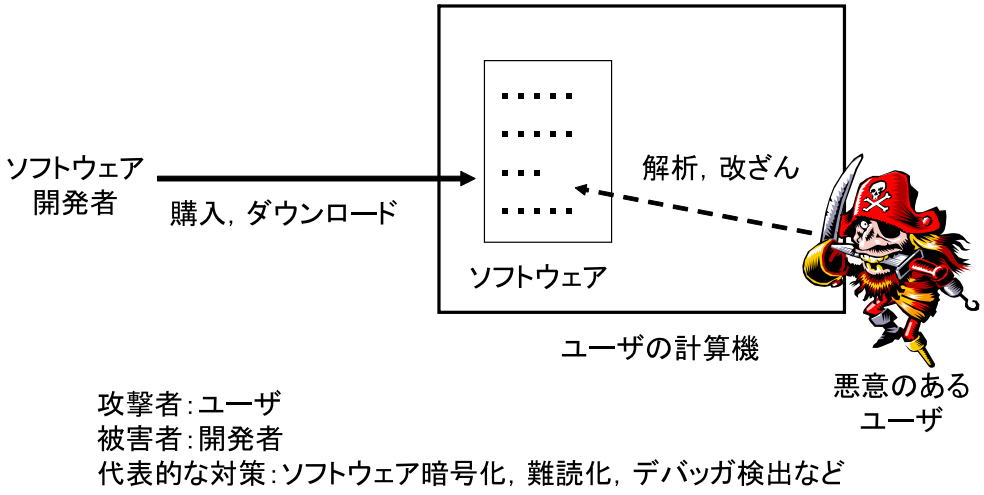


図 2.3 攻撃モデル 3

サブルーチン間や外部インタフェースの入出力を複雑にしておく必要がある．例えば，ライセンスチェックルーチンが，チェックの結果として 0 か 1 の値のみを返すプログラムは，攻撃が容易であり，対策が必要となる．1 つの対策は，返り値に乱数を混入させることである [35]．より高度なブラックボックス攻撃対策としては，暗号システムなどにおいては，セキュア API とその API を使うシステムとの間で，Secure Authenticated Channel (SAC) などが使用される．

2.2 ホワイトボックス攻撃

ホワイトボックス攻撃は，プログラム内部を解析する攻撃であり，Windows PC のようなオープンシステムでは，特に脅威となる．一般に，プログラム全体を読んで理解することは困難であるため，攻撃者は，デバッガや逆アセンブラを用いて攻撃の手がかり（特定の API 呼出し，文字列，命令，定数など）を探し，解読範囲を狭めて解析を行う．攻撃対策としては，プログラムを難読化する，攻撃の手がかりを隠す，攻撃ツールの動作を妨げる，攻撃を検知してプログラムを停止

する，などの方法がある．

2.3 サイドチャネル攻撃

サイドチャネル攻撃は，システム内部から漏れ出る情報（電圧の変化や入出力の時間変化など）を手がかりにして，システム内部の情報を得る攻撃である [59]．システムをハードウェアで実装した場合，ホワイトボックス攻撃は困難となるが，システムの仕様やアルゴリズムが既知である暗号システムにおいては，サイドチャネル攻撃が脅威となる．

3. ソフトウェア改ざん攻撃

攻撃者の目的の多くは，ソフトウェアを解析して秘密情報を得るのみならず，得られた情報に基づいてソフトウェアを改ざんし，改ざんしたソフトウェアを実行することにある．例えば，典型的な攻撃として，ソフトウェアを解析してライセンスチェックルーチンを発見した後に，ライセンスのチェックルーチンを削除したり，迂回するように改ざんし，実行することがよく行われる．

ソフトウェア開発者は，改ざんを検出する，もしくは，改ざんしたソフトウェアの実行を防止する仕組みを取り入れる必要がある．特に，1 箇所を改ざんしただけで攻撃が達成できてしまうことを “single point failure” と呼ばれており，何らかの対策が必要である．例えば，ライセンスチェックを 1 箇所でしか行わない場合がこれに該当する．

4. ツールによる攻撃

これまでに紹介した攻撃の多くは，何らかのツールを用いて行われる場合が多い．例えば，逆アセンブラ，逆コンパイラ，デバッガ，バイナリエディタ，メモリダンプツール，アンパッカー，モニタリングツール，ファイルスキャナなどがある [7]．

これらは，ソフトウェア開発に役立つ一方で，ソフトウェアのクラック行為にも悪用できる．これらの多くはインターネット上で容易に入手可能であり，ツールを用いてソフトウェアを解析する方法の解説書 [36][72] が数多く存在する．

ソフトウェア開発者は，何らかの方法によりこれらツールの実行を妨げることが求められる．

5. Code Injection 攻撃

Code Injection 攻撃は，ソフトウェアのセキュリティホールとなっているバッファオーバーフローを利用し，メモリ上のデータ領域を溢れさせてプログラムを書き込み，そこに実行を移すものである．Code Injection 攻撃は，ネットワーク経由の攻撃モデル 1(図 2.1) において特に脅威となるが，攻撃モデル 3(図 2.3) においても脅威となる．

多くの携帯端末機（ゲーム機，携帯電話など）では，プログラム認証機構を設けることで，非認証プログラムの実行を防止している．ところが，バッファーオーバーフローの脆弱性を利用して，保護の仕組みが破られるケースが後を絶たない．

例えば，SONY の PSP (Play Station Portable) は非署名プログラムが実行できないようになっているが，ゲームのソフトウェアの脆弱性を利用して，改変したセーブファイルをロードさせるなどにより，任意のコードを実行されたケースがある [69]．また，Apple の iPhone / iPod touch OS v1.1.1 の TIFF ライブラリの脆弱性を利用して，プロテクトを無効化し，任意のソフトウェアをインストールされたケースもある [70]．

バッファオーバーフローが発生するのは，特定のデータ構造に割り当てられたメモリの境界外にデータを書き込んだ時である．バッファオーバーフローの例を図 2.4 に示す．

この図では，あらかじめ確保されたメモリ領域に `gets()` 関数を使って文字列型配列データをコピーする．`gets()` 関数は，標準入力から文字をコピーし，EOF に達するか，改行文字を読み込むまでデータを格納する．この時，確保されたメモリ領域が 12 バイトの場合，文字列の終端に NULL が挿入されるため，11 バイ

トの文字列が格納できる．ただし，gets() によって 12 バイト以上の文字列配列が入力されると，あらかじめ確保されたメモリ領域を超えて，領域外のメモリまで書き込みが行われる．この状態をバッファオーバーフローと呼ぶ．

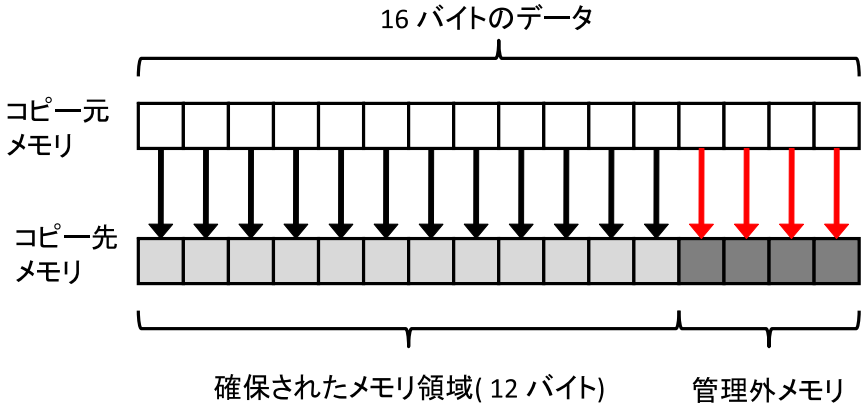


図 2.4 配列の境界を越えた書き込み

すべてのバッファオーバーフローがソフトウェアの脆弱性の原因となるわけではないが，もし攻撃者がユーザが制御できる入力を使ってセキュリティ上の欠陥を突く事ができるならば，バッファオーバーフローは脆弱性につながる可能性がある．

第3章　ソフトウェアプロテクションの要素技術

前章で紹介したようにソフトウェアに対する多様な攻撃方法が知られている一方で，数多くのプロテクション技術が提案されている．一般に，システムへの攻撃の対策は，prevention（攻撃の防止），detection（攻撃の検知），response（攻撃に対する応答）に分類される [48]．プログラムの難読化や暗号化といった技術は，prevention に属し，デバッガの検知やプログラム改ざんの検出は detection に属する．また，response は detection とセットになっており，detection により攻撃を検知した後，response においてシステムを停止することで攻撃を無効化するのが一般的である．一般に，あらゆる攻撃を完全に防げる prevention 技術は存在しないため，detection および response 技術と組み合わせてシステムを保護することが必要である．

prevention 技術には，ソフトウェア難読化，暗号化，逆アセンブルの防止などがある．また，detection & response 技術としては，Integrity Verification やデバッガの検出などがある．以降，各要素技術について説明する．

1. ソフトウェアの難読化

1.1 名前難読化

名前難読化ツール

名前難読化は，ソースコードを対象にソースコード中の変数，関数，メソッドなどの名前を書き換える事である．名前難読化により，各変数，関数，メソッドの役

割を推測しにくくなる．市販されているツールとして，Dotfuscator, Spices.NET, DashO などがあり，そのほかにも Web で公開されている難読化ツールが多数存在する．しかし，これら変数，関数，メソッド名を隠しても秘密を隠していることにはならない．

オーバーロードインダクション

オーバーロードインダクションは，名前難読化の一種であり，オブジェクト指向言語におけるメソッドの多重定義（オーバーロード）を利用して，多数のメソッド名を可能な限り重複させる方法である [67]．これにより，プログラム中のメソッドの約 1/3 を “a()” という名前にすることができ，メソッドの定義と呼び出しの関係を解析しにくくできる．

動的名前解決

動的名前解決は，ソースコード中に現れる関数やメソッド名を隠す名前難読化手法である．通常，ライブラリ関数（メソッド）は外部参照を行うため，これらの名前を隠す事は困難である．玉田らの手法は，ソースコード中に存在する関数，およびメソッド名を暗号化しておき，動的名前解決を用いてプログラム実行中に暗号化しておいた名前を復号し，復号した名前の関数を動的に実行する方法である [62][63]．この手法は，通常隠す事が難しいライブラリ関数（メソッド）の呼び出しを静的に隠蔽できるのが特徴である．ただし，スタックトレースなどの動的解析には弱い．

1.2 制御フローの難読化

制御フローの変換

制御フローの難読化は，プログラムの制御フローを変換することによって，攻撃者に制御フローの把握を困難にするといった手法である．門田らは反復構造を持つプログラムを実行効率を落とさずに難読化する手法を提案している [50]．

条件分岐文の挿入

Collberg らは，恒真あるいは，恒偽な恒等式を Opaque predicate として利用し，分岐文を挿入することで制御フローの難読化する手法を提案している [16]．図 3.1 は，Opaque predicate を用いた分岐文の挿入の例である．左のフローチャートがオリジナルのプログラム，右のフローチャートが Opaque predicate による条件分岐文が挿入され，難読化されたプログラムである．

このプログラムは，条件分岐 E が真ならば，$S1 \rightarrow S2 \rightarrow S3 \rightarrow S4$，偽ならば，$S1 \rightarrow S2$ の基本ブロックを通る．このプログラムに Opaque predicate な条件分岐文 P^F と Q^F を挿入し，制御フローを複雑にする．ただし，P^F と Q^F は恒偽のため，制御フローが複雑になっても条件分岐 E の箇所しか，分岐が起こらないので，オリジナルのプログラムと動作が変わらない．

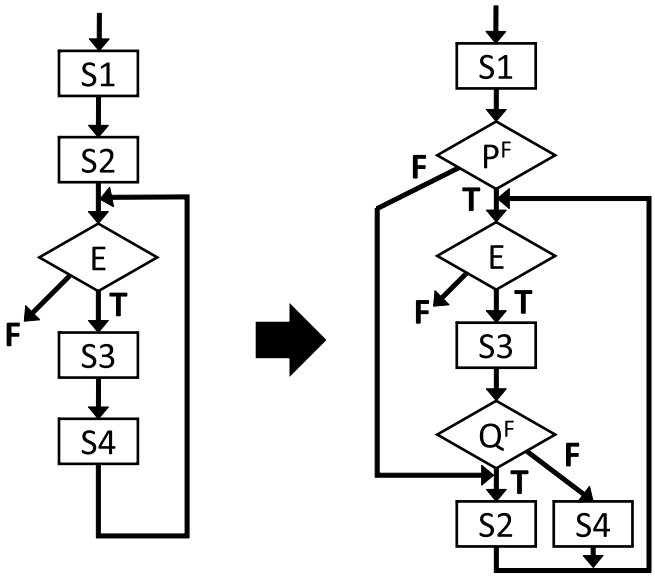


図 3.1 Opaque predicates による条件分岐文の挿入

制御フローの平坦化

Wang らは，分割・平坦化された各基本ブロックの実行制御に変数ポインタを使うことで，静的解析による制御フローの復元（難読化の逆変換）を困難にしている [71]．制御フローの平坦化の例を図 3.2 に示す．左の図は，基本ブロック $B1$ から $B5$ が $B1 \rightarrow B2 \rightarrow B3 \rightarrow B4 \rightarrow B5$ へと逐次処理する．この処理に変数ポインタ i を導入し，switch 句を用いて制御フロー変換する．

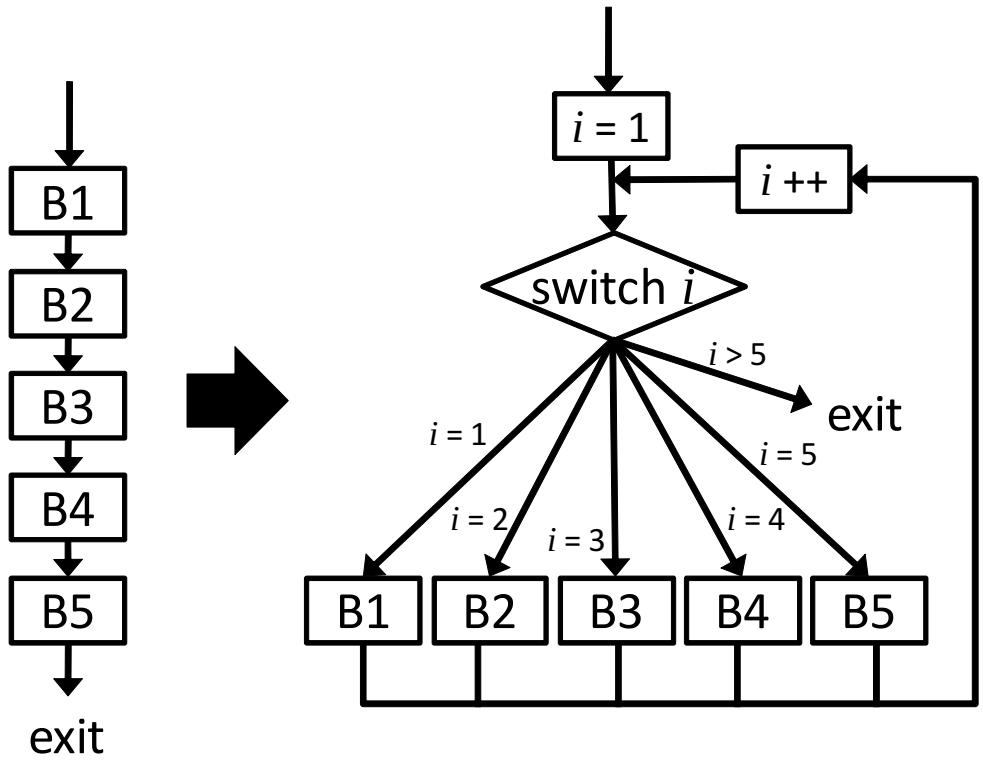


図 3.2 制御フローの平坦化

変数の付け替え

Chow らは，Wang らの手法に加え，複数の基本ブロックを融合させ，ダミーステートメントを混在させた Emulative lump と呼ばれる基本ブロック (図 3.3 の

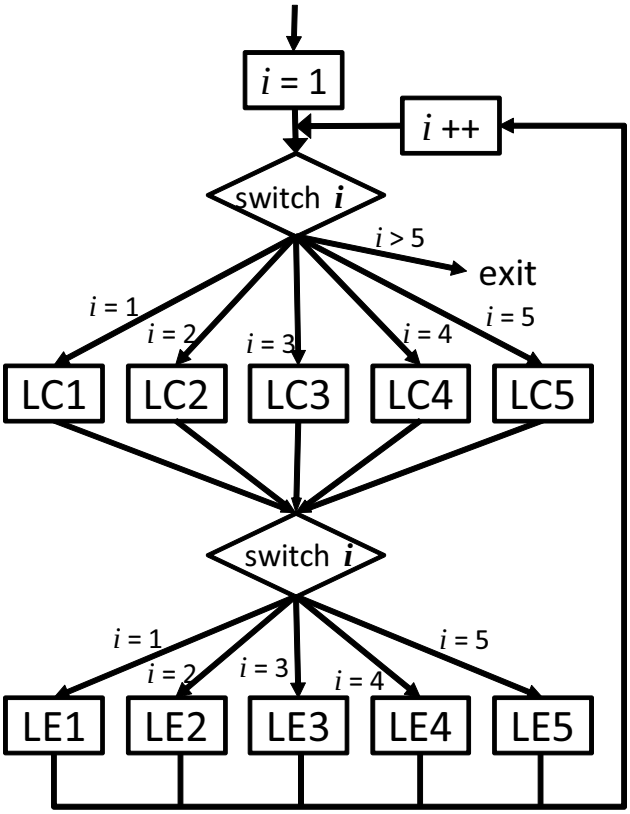


図 3.3 変数の付け替え

$LE1$ から $LE5$) を生成し，さらに Connective lump と呼ばれる変数変換テーブル (図 3.3 の $LC1$ から $LC5$) を導入する事で，各 Emulative lump の動作を解析困難にしている [13].

1.3 モジュール間呼び出し関係の難読化

Ogiso らは，関数モジュール間呼び出し関係を関数ポインタの付け替えによって実現し，モジュール間の呼び出し関係を複雑にして難読化する方法を提案している [52]．一般に，ポインタの指すオブジェクトの特定 (point-to analysis) は NP 困難であるため，この難読化方法は静的解析に対して効果を発揮する．

1.4 データ構造の難読化

Collberg らは文献 [17] において，データ構造を変換する難読化を提案している．例えば，1 つの配列を 2 つの配列に分割したり，2 つの配列を 1 つの配列に統合したり，一次元配列を多次元配列に変換したり，多次元配列を一次元配列に変換する方法が提案されている．これらの方法は，難読化の効果はあまり高くないが，実施が容易である．

1.5 データ値の難読化

文献 [6][56] において，準同型写像を用いてデータ値を変換する難読化が提案されている．これらを用いる事で，暗号の復号鍵や署名生成のハッシュ値などが計算過程中にメモリ内に露出することなく計算が可能になる．

1.6 高レベル命令から複数の低レベル命令への置換

Mambo らは，バイナリプログラム中のアセンブラ言語の高レベル命令に対して複数の低レベル命令への置き換え，命令の入れ替え，ダミー命令の挿入を行う事で，バイナリプログラムの解析を困難にする方法を提案している [44]．

1.7 コードの自己書き換え

Kanzaki らは，コードの自己書き換えを用いて，あらかじめバイナリプログラムの命令をダミー命令に書き換え，実行時にそれらの命令を正しい命令に置き換える事で攻撃者がバイナリプログラムに対する静的解析を失敗させる保護方法を提案している [34]．

2. White-box Cryptography

暗号プログラムを保護する最も強力な難読化手法の 1 つに，Chow らよって提案された White-box Cryptography [11][12] がある．この提案以降，この方法は盛

んに研究されるようになった[33][40]．この方法では，復号鍵を使用する複数の計算をテーブル参照に置き換え，それにより,（復号）鍵はテーブル内部に隠され，攻撃者に認知されないようになる．しかし，この手法は特定の暗号プログラムに強い保護を付与することを可能とするが，大きなメモリ領域（数メガバイト）を必要とし，深刻な実行速度の低下などのペナルティがあるため，その適用範囲は限られている．したがって，モバイル端末のような計算機資源に制限のある環境については，攻撃者から暗号プログラムを保護するために別の手法を必要とする．

3. ソフトウェアの暗号化

ソフトウェアの暗号化は，ソフトウェアを実行できる形式のまま暗号化する方法である．動作原理は，対象となる実行ファイルを暗号化（または圧縮）して，プログラムの先頭に復号（または展開）ルーチンを付加することで実現する．これを実現するツールをパッカーと呼び，多数のパッカーが公開されている [7].

4. 逆アセンブル対策

逆アセンブル対策として，バイナリプログラム中に逆アセンブラが解釈できないような数値を挿入する方法がよく知られている．一部のデバッガでは，recursive traversal というより高度な逆アセンブルアルゴリズムが採用されているが，各分岐命令のジャンプ先を動的に決定する機構を導入することで対策できる [41].

5. Integrity Verification

Integrity Verification とは，完全性検証をいい，ソフトウェアの完全性とは，ユーザが使用するソフトウェア自身やその設定ログなどに改ざんがなくオリジナルの状態である事を指す．既存技術としては，ハッシュ値の利用やファイル整合性チェックソフト，システムスキャナがある．

5.1 侵入検知システム

侵入検知システム (IDS) とは，ネットワーク，サーバなどの機器やアプリケーションをモニタリングし，不正なアクセスや挙動を検出し，記録するシステムである．IDS は，主に 2 種類ある．1 つは，ネットワーク上に設置して，異常なトラフィックや不正アクセスの疑いを管理者に通知するネットワーク型 IDS (NIDS)，もう 1 つは，サーバマシンにソフトウェアとして組み込まれ，対象のサーバに異常が発生していないかを監視する．異常が確認された際には，NIDS と同様に通知を行うホスト型 IDS (HIDS) がある．HIDS は，サーバにソフトウェアとして組み込まれ，サーバ上のファイルシステムの監視，および，変更の検知が可能である．

本論文が対象とする攻撃モデル 3(図 2.3) では，ネットワーク経由の攻撃は対象外であるが，HIDS におけるファイルシステム変更の監視，検知の仕組みは攻撃防止に役立つ可能性がある．ただし，監視や検知の仕組みを担うプログラムをエンドユーザから隠蔽する（アクセスコントロールなどにより）ことが前提となる．

6. デバッガの検出

デバッガを検出する方法は様々な方法があり，デバッグ用の関数，API，割り込みやデバッガの挙動によって，デバッガの検出ができる．主な検出方法としては，次のような手法が挙げられる．

1. IsDebuggerPresent 関数
2. 仮想デバイスドライバ検出
3. ウィンドウクラス名取得
4. INT3 サーチ
5. 特定メモリエリアのアクセス属性取得
6. 構造化例外ハンドラを用いたデバッグ検出

7. GetThreadContext 関数を用いたハードウェアブレークポイント検出
8. ネイティブ API の NtQueryInformationProcess 関数を使用したデバッグ検出
9. INT 命令によるデバッガ検出
10. 実行時間計測

7. プログラムの認証

多くの携帯端末機（ゲーム機，携帯電話など）では，電子署名などを用いて，プログラムの認証機構を設けることで，非認証プログラムの実行を防止している．この機構により，改造されたプログラムのインストールや実行を防止できる．

8. ハードウェアを併用したソフトウェア保護

ソフトウェア単体でのプロテクション技術は，White-box 暗号システムなどの有力な方法があるが，その有効範囲は限定されている．例えば，ライセンスチェックルーチンのプログラム中の機能分岐点を完全に隠蔽する事は事実上不可能である．ライセンス認証に成功した場合，および，失敗した場合の計算機の挙動や状態を測定し，比較する事でそれらの差分から機能分岐点が特定できるためである．このような動的解析からプログラムを保護するために，計算機システム上のメモリやバスが情報を漏らさないようにブラックボックス化する必要がある．

8.1 セキュアプロセッサ

セキュアプロセッサを用いたアプローチでは，メモリやバスが情報を漏らさないようにブラックボックス化する．このことは，メインメモリ上の暗号化されたプログラム及びデータを，CPU 内部において復号しながら実行することにより実現される．復号後の命令系列は，CPU 内部のキャッシュメモリに展開される．

このようなプロセッサ内部のキャッシュメモリを “仮想的なブラックボックス” として用いる方法は数多く提案されており，Cerium[9]，XOM[39]，AEGIS[60] などのアーキテクチャが提案されている．

この方法では，外部メモリ上のデータやプログラムを暗号化しておき，それらが必要になった場合にのみキャッシュメモリへと復号し，実行する．プロセッサチップは物理的・電気的に耐タンパー性を持つことが要求される．CPU がプロセッサ外部のメモリにデータの書き込みを行う場合には，書き込み前にデータを暗号化する．

この方式の欠点は，チップ外部とのデータ入出力時に暗号化/復号処理を必要とするため，実行パフォーマンスが低下することである．一般に，暗号ブロック長を大きくする，または，強度のある暗号方式を採用した場合には，この低下はより深刻なものとなる．実行効率の観点から，対称鍵暗号を用いることになるが，安全性の面から，全暗号ブロックで同一鍵を用いるわけにはいかない．そこで，マスター鍵となる情報，各暗号ブロックが記録されている外部メモリのアドレス，及び，ブロック内の特定のビット列などからブロック毎に個別の鍵を生成し，復号することとなる．

8.2 アクセス制御による仮想ブラックボックス

仮想的なブラックボックスを計算機内に構築する一つの手段として，プロセッサ外部の RAM に強力なアクセス制御を行う方法がある（Microsoft 社は，ブラックボックス化されたメモリ領域のことを curtained memory と呼んでいる）．

ただし，この方式を実現することは容易ではない．今日の多くのオペレーティングシステム (OS) では，各アプリケーションが占有するメモリに対し，互いにアクセスできないようにアクセス制御を行い，不正なプログラムから各プロセスを保護している．しかし，バッファオーバーフローを利用した攻撃の問題，OS のアクセス制御機構自身が改ざんされるという問題，OS の欠陥によるセキュリティホールの問題，デバッガの存在，DMA (Direct Memory Access) デバイスの存在といった懸案事項があり，従来の PC 環境で堅牢なアクセス制御を実現することは困難であった．

この方式を実現するためには，ブラックボックス実行のための専用の実行モードを持ったCPUや，キーボード，マウス，ビデオその他のデバイスにセキュアI/O機能（入出力データの暗号化）を持たせることが必要である．また，OSが正当なものである（改ざんされていない）ことを保障するための特別なブートストラップ方式 が必要である．例えば，TCG (Trusted Computing Group) 準拠のシステムでは，Trusted Platform Module (TPM) と呼ばれる耐タンパーハードウェアが，システムブート時に各ソフトウェアコンポーネント（BIOS，ブートローダ，OS）のハッシュ値を計測し，保存する．これらのハッシュ値により，各コンポーネントの改ざんが順次検出できる．なお，TPM自身もEndorsement Keyと呼ばれる鍵を含んでおり，任意のコンポーネントに対してTPMの正当性を示すことができる．アプリケーションの開発者は，特定のTPMやOS上でのみ動作するよう，アプリケーションに署名をつけることができる．

8.3 耐タンパーハードウェア

耐タンパーハードウェアとは，許可されていない変更を行うと正常に動作しないハードウェアをいう．文献[25]は，暗号モジュールを4つセキュリティレベルに分け，それぞれのレベルにおいて満たすべきセキュリティ要求が記述されている．

耐タンパー性に関連する事項としては，暗号モジュールのポート及びインタフェース，認証によるアクセス制御，物理的セキュリティ，暗号鍵管理，電磁妨害/電磁両立性 (EMI/EMC) が該当する．

9. バッファオーバーフロー対策

9.1 開発者によるソースコードレビュー

バッファオーバーフローを予防する方法は，開発者が次のようなプログラミングルールを実践することができれば理想である．

- データをバッファに書きだす前には，常に配列の境界を確認する．

- 入力文字の数やそのフォーマットを制限する関数を使用する．
- `scanf()`, `strcat()`, `getwd()`, `gets()`, `strcmp()`, `sprintf()` などのバッファオーバーフローを引き起こすリスクが高い関数の使用を避け，安全なライブラリや関数を使用する．

9.2 ツールによる自動検査

関数名照合型検査ツール

ソースコードから，9.1 節中に挙げたバッファオーバーフローを引き起こすリスクが高い関数名を警告する検査ツールがある．Rats，Flawfinder，ITS4 などが無償で提供されている．

構文解析による自動検査

ソースコードを構文解析によって，抽象化した構文木に変換する事でソースコード内部の論理的な矛盾点を発見する事ができる．商用のツールとしては，MathWorks 社の PolySpace，FORTIFY 社の FORTIFY SCA，Coverity 社の Coverity Static Analysis などがある．

9.3 コンパイラのアドオンツール

近年では，C/C++ において，実行時のメモリ境界検査を実施するためのさまざまなコンパイラのアドオンツールやライブラリが多数提供されている．

StackGuard

StackGuard[22] はスタック上の戻りアドレスを更新から保護することにより，スタック破壊攻撃を検出する．関数が呼び出された時に，戻りアドレスの前にカナリア (canary) と呼ばれる値を設定し，バッファーオーバーフロー攻撃を受けた

時に StackGuard は，関数が戻る時にカナリアが変更されている事を検出し，警告をログに出力してプログラムを終了する．

具体的には，GCC のコードジェネレータの関数 `function_prolog()` と `function_epilog()` に対するパッチとして実装されており，関数の実行開始時には，`function_prolog()` においてカナリアを挿入し，関数終了時には，`function_epilog()` においてカナリアの検証を行う．この方法により，戻りアドレスを崩壊させるような攻撃は，関数が戻る前に検出することができる．

SSP (ProPolice)

StackGuard から派生した中でもよく使われている GCC の SSP (Stack Smashing Protector) があり，ProPolice とも呼ばれる [24]．SSP は GCC の中間言語トランスレータとして実装され，C 言語で書かれたアプリケーションを一般的なバッファオーバーフロー攻撃から保護する．SSP も StackGuard 同様にガード（カナリア）により，引数や戻りアドレス，以前のフレームポインタに対する変更を防止する．保護対象の関数のソースコードが与えられると，プリプロセッサによって，図 3.4 に示すようなコードが適切な位置に挿入される．カナリアはアプリケーションの初期段階で生成された乱数が設定され，権限のないユーザがそれを知る事を防いでいる．

この機能は，GCC のオプションを有効にする事で使用できる．例えば，`-fstack-protector` オプションと `-fno-stack-protector` オプションによってスタック破壊保護の有無を切り替える事ができる．また，`-fstack-protector-all` オプションと `-fno-stack-protector-all` オプションは，文字配列を伴う関数だけでなく，全ての関数を保護するかどうかを指定できる．

Visual C++ の /GS オプション

Visual C++ では，コンパイルオプション /GS が SSP と同様の機能を持つ．ただし，Visual C++ では，カナリアのことをクッキーと呼ぶ．SSP のカナリア

- ローカル変数の宣言

```
volarile int guard; /* カナリア */
```

- 入力地点

```
guard = guard_values;
```

- 終了地点

```
if (guard != guard_values) {
 /* エラーログを出力 */
 /* 実行中止 */
}
```

図 3.4 SSP (ProPolice) によるカナリアの挿入

との違いは，クッキーは関数呼び出し毎に異なる値をとり，スタック破壊攻撃を試みる攻撃者はカナリアの予測困難にしている．

Visual C++ の /RTC オプション

Visual C++ は，スタックの実行時検査の仕組みを提供し，スタックポインタの破損やローカル配列の超過書き込みなどの一般的な実行時エラーを捕捉する事ができる．なお，/RTC オプションは，デバッグビルド時のみ有効である．

9.4 その他の技術

バッファオーバーフロー対策に有効なその他の技術としては，アドレス空間のランダム化，スタック，ページ，セグメントの非実行化などが知られている．

第4章　段階的ソフトウェアプロテクション

1．攻撃と防御の対応

2 章で述べた攻撃方法に対し，3 章のプロテクション技術がどのように対応するかを下記に示す.

攻撃者の目的の多くは，ソフトウェアを解析して秘密情報を得るのみならず，得られた情報に基づいてソフトウェアを改ざんし，改ざんしたソフトウェアを実行することにある．特に，ソフトウェア中の 1 箇所を改ざんしただけで攻撃が達成できてしまうケースは “single point failure” と呼ばれ [8]，対策が必須となる．

一方，あらかじめ何らかの攻撃対策がなされているシステムに対しては，バッファオーバーフローを利用した Code Injection 攻撃 [27] によってその対策が破られることが脅威となる．多くの携帯端末機（ゲーム機，携帯電話など）では，プログラム認証機構を設けることで，非認証プログラムの実行を防止している．ところが，バッファオーバーフローの脆弱性を利用して，保護の仕組みが破られるケースが後を絶たない [69].

これらの攻撃に対する可能な対策は，およそ表 4.1 のように整理される．まず，ブラックボックス攻撃対策には，システムの外部インタフェースとして Secure Authenticated Channel (SAC) を用いたり，内部インタフェースとしてセキュア API[21] を用いるたりすることが有力である．ただし，いずれの方法も，ホワイトボックス攻撃が防止できていることが前提となっている．一般に，完全にホワイトボックス攻撃を防止することは困難なことから，外部インタフェース，内部インタフェースともに，なるべく複雑化して仕様の推測を困難にすることが重要となる．また，有力なブラックボックス攻撃の 1 つにリプレイ攻撃 [61] が知られ

表 4.1 攻撃タイプとプロテクション技術の対応

攻撃のタイプ		対応するプロテクション技術
解析	ブラックボックス	SAC, セキュア API[21]
	ホワイトボックス	難読化 [20], 暗号化 [7], White-box Cryptography[11], セキュアプロセッサ [60]
	サイドチャネル	アルゴリズム変換 [26], 耐タンパーハードウェア [25]
改ざん		暗号化，セキュアプロセッサ
改ざん後のプログラムの実行		Integrity Verification[55]，プログラムの認証 [23]，TPM[66]
Code Injection		データ実行防止，メモリスキャン，Software Diversity[31]
攻撃ツールの利用		アンチデバッグ技術 [7]，逆アセンブル対策 [41]

ており，少なくとも リプレイ攻撃 が困難なプロトコルの採用が必要である．

ホワイトボックス攻撃対策としては，暗号化 [7] によってプログラムを読めなくしたり，難読化 [20] によってプログラムの理解を困難にしたりする方法が有力である．ただし，暗号化したプログラムは実行時に復号されるため，復号後のプログラムを解析される恐れがある．また，多くの難読化手法は静的解析に効果を発揮するが，動的解析に対しては効果が弱い．難読化と暗号化については文献 [20] が網羅的で詳しいので参照されたい．White-box cryptography[11] は，適用対象が限られるが，鍵内蔵型ソフトウェアにおいて鍵を隠蔽するには非常に強力である．ただし，大きなメモリ領域を必要とし，実行速度の低下が大きいという課題がある．根本的な対策としては，セキュアプロセッサ（[60] など）を用いる方法がある．多くのセキュアプロセッサでは，暗号化されたプログラムを CPU キャッシュ内などの解析不能なエリアで復号して実行することで，プログラムの解析を防いでいる．ただし，汎用 CPU が使えないために大きなコストを要する．

サイドチャネル攻撃対策は，既知の暗号アルゴリズムを採用する際に必要とな

る．対策として，アルゴリズムを（少し異なるものへと）変換する方法が知られている [26]．また，物理的，及び，電気的な耐タンパーメカニズムを施すことが望ましい [25].

改ざん攻撃に対しては，暗号化によって，プログラム実行前の改ざんを防ぐことが望ましい．ただし，プログラム実行前には復号されることから，復号後のプログラムを改ざんされる恐れがある．可能であれば，より強力なプロテクションが得られる，セキュアプロセッサの採用が望ましい.

改ざん後のプログラムの実行を防ぐ方法としては，プログラム実行開始時に，プログラムが改ざんされていないことを検査する Integrity Verification[55] や，電子署名を利用したプログラム認証がある．その具体的な方法として，Trusted Computing Group[66] の策定する Trusted Platform Module (TPM) を用いて，システムブート時に各ソフトウェアコンポーネントの改ざんを検出する方法がある．ただし，Code Injection 攻撃によって破られる恐れがある [5] ため，改ざんによる攻撃達成が困難なプログラム（例えば，single point failure が存在しないなど）を作成することが先決となる．なお，認証されたプログラムの実行開始後は，一般に，改ざんチェックを行うことは難しい.

Code Injection 攻撃に対しては，CPU の NX (No eXecute) ビットや OS のデータ実行防止機能（Microsoft Windows における Data Execution Prevention 機能など）を用いてメモリ上のデータセグメントとプログラムセグメントを分離する方法が有力である．ただし，データ実行防止機能を制御するプログラムが攻撃され，無効化される場合があるため，バッファオーバーフローそのものが生じないようにプログラムの脆弱性を除去することが先決である．また，システム全体のパフォーマンスが一時的に低下することを許容するならば，ウィルス検査ツールで採用されている定期的なメモリスキャンも有力な方法である．そのほか，ソフトウェアに多様性を持たせる (Software Diversity) という考え方に基づいて，命令セット，データ格納アドレス，ライブラリのロード順序などのランダム化を行うという対策も有力である [4][31].

2. 段階的ソフトウェアプロテクション

前章の議論から，多様な攻撃からソフトウェアを保護するためには，TPM などのハードウェアセキュリティチップやセキュアプロセッサによるシステム設計レベルの対策を行うのみならず，プログラムの実装レベルのプロテクション技術（難読化，逆アセンブル防止など），White-box Cryptography などによるアルゴリズムレベルのプロテクション，外部設計レベルのプロテクション（外部インタフェースの複雑化）などを併用することが望ましい．これらを，ソフトウェアの開発工程ごとにまとめたものを表 4.2 に示す．以降，表 4.2 の各開発工程におけるプロテクション技術について説明する．

本節では，ソフトウェア開発の各段階において，それぞれどのような対策を行う必要があるかを整理し,「段階的ソフトウェアプロテクション」としてまとめる．例えば，ソフトウェア開発者は，ソフトウェア外部仕様の策定段階において，インタフェースの仕様を注意深く決めることでブラックボックス攻撃を防ぐことがまず必要である．また，ソフトウェア内部仕様の策定段階では，single point failure が発生し得る箇所について，対策を講じる必要がある．さらに，ソフトウェアの実装段階では，難読化や暗号化により，ソフトウェアの実装のレベルでホワイトボックス攻撃を防ぐ必要がある．このように，段階的ソフトウェアプロテクションでは，開発の各工程において段階的にプロテクション技術を適用する．

2.1 プログラムの実行制御

個々のプログラムの保護方法について検討する前に，プログラムの実行環境にプロテクション機構を導入することが望ましい．すなわち，開発者が認めたプログラムのみ実行を許可し，改ざんされたプログラム，もしくは，エンドユーザが独自に作成したプログラムを動作させないような仕組みが必要である．その 1 つの手段が，プログラムの認証であり，プログラムに対する署名の方法，署名鍵の配布方法，秘密鍵の格納方法などを決定する必要がある．

表 4.2 段階的ソフトウェアプロテクション

開発工程	プロテクション技術
システム設計	TPM, プログラム認証，メモリスキャン, Software Diversity, データ実行防止
S/W 外部設計	SAC, 外部仕様の複雑化，ランダム性の導入，Replay attack 対策
S/W 内部設計	セキュア API, single point failure 対策
アルゴリズム設計	アルゴリズム変換
実装	脆弱性の解消，ソースコード難読化，バイナリコード難読化，逆アセンブル対策，ソフトウェアの暗号化，動的解析対策
実装後	デバッガ検出

2.2 外部仕様のプロテクション

保護対象のソフトウェアの外部設計工程において，ブラックボックス攻撃の対策として，ソフトウェアの外部仕様である入出力の弱点を解消する必要がある．一般に，入出力が単純であると攻撃されやすい．たとえば，ライセンスチェックルーチンの戻り値が 0 か 1 のいずれかの値を取り，かつ，0 がライセンスあり，1 がライセンスなしを表すような場合，常に 0 を返すように改ざんされる恐れがある．戻り値の値域を大きしくて仕様を複雑化するとともに，値にランダム性を持たせるなどにより リプレイ攻撃を防ぐことが必要となる．システムの外部機器やネットワーク経由のアクセスには，SAC などを用いることも検討することが望ましい．

2.3 内部仕様のプロテクション

ソフトウェアの内部設計では，single point failure の対策をまず行う必要がある．例えば，ライセンスチェックの判定が 1 箇所で行われる場合，その判定箇所

を削除する，または，迂回することでライセンスチェックが無効化される恐れがある．このような場合，複数の場所でチェックする [8]，もしくは，チェックの手続きを複雑にするなど内部設計の段階での対策が必要となる．また，可能であれば内部インタフェースとしてセキュア API を用いることが望ましい．

2.4 アルゴリズムのプロテクション

公開されている暗号方式を採用する場合など，アルゴリズムが既知である場合には，ホワイトボックス攻撃とサイドチャネル攻撃の対策として，アルゴリズムそのものを変換して，攻撃耐性を高めることが望ましい．アルゴリズムによっては White-box Cryptography を用いることが考えられる．また，データの難読化，ループの難読化などの実装レベルのプロテクションを適用することで，結果的にアルゴリズムを同定されにくくできる [75].

2.5 実装レベルのプロテクション

脆弱性の解消

プログラム認証の仕組みを導入した場合，プログラム実行開始時点において，非認証プログラム（改ざんしたプログラムなど）の実行を防ぐことができるが，バッファーオーバーフローの脆弱性があると，保護の仕組みが破られる場合がある．そこで，プログラム実装中の脆弱性の有無を調査し，あらかじめ修正しておくことが必要となる．

ソースコードの難読化

ホワイトボックス攻撃対策として，プログラムに含まれる攻撃の手がかりとなる情報を隠蔽する．攻撃者は，特定の API 呼び出し，外部出力，特徴的な演算・データ型など，攻撃の手がかりとなる情報を探索し，ソフトウェアの解析を進める．開発者は，多数の難読化手法を適材適所に用いて攻撃の手がかりを隠蔽する必要がある [20][75].

バイナリプログラムの難読化

コンパイル後のバイナリプログラムに対して，高レベル命令から複数の低レベル命令への置換 [44]，コードの自己書き換え [34] といった難読化方法を用いて，ホワイトボックス攻撃を困難にする．

逆アセンブル対策

逆アセンブル対策として，バイナリプログラム中に逆アセンブラが解釈できないような数値を挿入する方法がよく知られている．一部のデバッガでは，recursive traversal というより高度な逆アセンブルアルゴリズムが採用されているが，各分岐命令のジャンプ先を動的に決定する機構を導入することで対策できる [41]．

ソフトウェアの暗号化

ソフトウェアを暗号化することによって，解析コストを増大させる．数多くのソフトウェア暗号化ツールが販売されている [7]．

動的解析への対策

多くの暗号化，難読化手法は，動的解析に対して効果が弱い．そこで，動的解析対策として，プログラムの挙動（命令の実行系列，データへのアクセスなど）に何らかのランダム性を持たせることが望ましい [2]．

2.6 追加のプロテクション

プログラムの実装終了後，追加のプロテクションとして，デバッガのインストールや動作を検知し，自らの実行を停止させたり誤動作させたりする機構を導入する [7]．近年では，デバッガではなく，Virtual Machine 上で攻撃対象のプログラムを動作させて解析を行うこともあることから，Virtual Machine を検知することも必要である．

改ざん検出機構の導入

ソフトウェアが改ざんされた際に，ソフトウェア自身で改ざんを検出し，実行を停止する（もしくは誤動作する）といった機構を導入する．

解析ツールの動作を妨げる仕組みの導入

デバッガのインストールや動作を検知し，自らの実行を停止させたり誤動作させることで解析を妨げる．

3. まとめ

ソフトウェア開発の各工程において段階的にプロテクション技術を適用することで，外部仕様，内部仕様，アルゴリズム，実装といった，ソフトウェアの各抽象化レベルにおける弱点を防御することが可能となる．また，ソフトウェア開発開始前の段階において，プログラムの実行制御の仕組みを導入したり，開発終了後の段階において改ざん検出機構やデバッガ検出機構を導入することで，2 重，3 重に防御を行うことが可能となる．

ただし，実装レベルのプロテクションにおいて，ソースコードの難読化は非常に数多く提案されているにも関わらず，それらを適材適所に用いる方法は従来明らかでない．次章では，多様な難読化手法を適材適所に適用するための枠組み（難読化フレームワーク）を提案する．

第5章　ソフトウェア難読化フレームワーク

1. はじめに

第 3 章において紹介した難読化技術は，それぞれ任意のプログラムに適用可能であるが，攻撃者のゴールを達成するために行動を妨げるというより,「複雑なプログラムの構築」を目的としている．これら難読化手法の多くは軽量，すなわち計算機資源の限られた環境にも適用可能であり，実行速度の低下が少ないという利点がある．一方で，プログラム内部の秘密情報を保護する目的に対して，どの程度有効かは不明確である．なぜなら，プログラムが複雑であるからといって，秘密情報を獲得するための攻撃が困難になっているとは限らないためである．さらに，難読化手法の多くは，明確な脅威モデル（セキュリティを破壊する攻撃者の振舞いのモデル）を仮定しておらず，どのような攻撃に対して有効であるのかが不明である．

一方，White-box Cryptography のようなアルゴリズムレベルの強力な難読化手法も存在するが，この技術は適用対象が特定のアルゴリズムに限定されており，本章の議論からは除外する．

本章では，攻撃者の能力を明確にし，プログラムの解析の際に行う攻撃者の行動（タスク）によって，それぞれの行動を妨げるように複数の難読化手法を選択する枠組みを提案する．

2. ゴール指向分析よるソフトウェア難読化の適用

2.1 既存の難読化の問題

3 章において，ソフトウェアプロテクションの要素技術としてソフトウェアの難読化について述べた．ただし，これら難読化を「どこに」「どのように」保護対象のプログラムに適用するかといった系統的な難読化の適用については議論はされていない．

したがって，既存の難読化の問題としては次の 3 つの問題が挙げられる．

問題 1: **保護対象のプログラムのどの部分**に難読化を適用すればよいのか．

問題 2: 保護対象のプログラムに**どのような難読化を適用**すればよいのか．

問題 3: 保護対象のプログラムが**どの程度保護された**のか．

これらの問題を解決するため，次節以降，系統的な難読化の適用のフレームワークについて述べる．

2.2 基本アイデア

図 5.1 は，ソフトウェア難読化の従来のアプローチと，提案するアプローチの概念を示す．図 5.1(a) は，従来のアプローチであり，入力としてプログラム P をとり，難読化されたプログラム P' を得るために一般的な難読化 $O(x)$ を導出する．このアプローチでは，$O(x)$ は想定する攻撃者から既知の P に含まれた秘密情報の保護に責任を負うべきである．しかしながら，従来の難読化手法のほとんどは攻撃者の攻撃に対して十分に考慮がされていない．一方，図 5.1(b) の提案するアプローチでは，P と想定する攻撃者モデル (2.4 節) を入力として，ゴール木を生成するためにゴール指向分析を行う．ゴール木は，攻撃者が達成するゴールとそのサブゴールによって形成される．その後，提案手法では，ゴール木中の分解した全てのサブゴールに対して，適切な既存の難読化手法 $O_1(x), \ldots, O_n(x)$ を適用する．過去の論文において，攻撃者の解析行動を破たんさせる目的に既存の難読

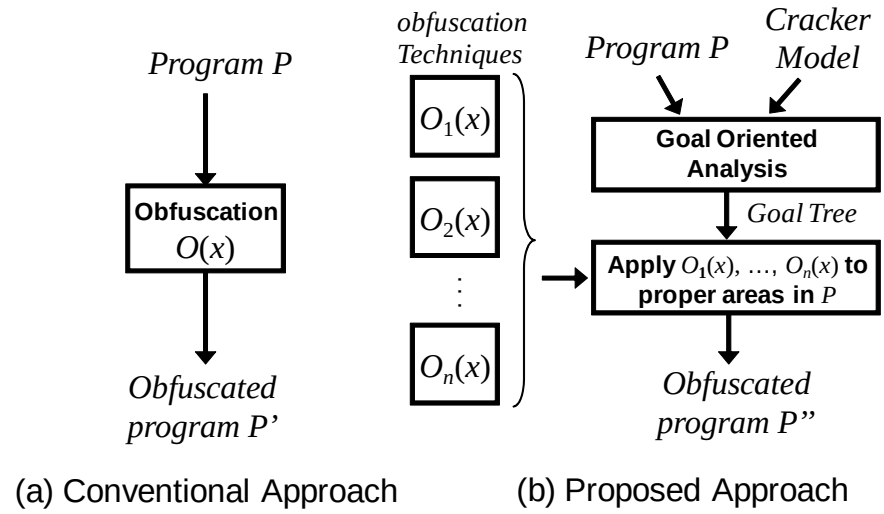


図 5.1 提案手法のコンセプト

化手法を用いるためのガイドラインを提案した [74]．しかし，文献 [74] では難読化手法の系統的な適用については議論をしなかった．

そこで本章では，系統的にプログラム中の秘密情報を保護するため，ゴール指向のアプローチを提案する．キーアイデアは，プログラム P における解析の目的と目標を持った想像上の攻撃者を仮定し，ゴールをいくつかのサブゴールに分解し，それぞれのサブゴールに対して，適切な難読化を適用する．

まず攻撃者の能力を決定する．次に，攻撃者の攻撃ゴールを定め，ゴール指向分析を実施する．ゴール指向分析によって，攻撃者のゴールをより具体的なサブゴールに分解（および，洗練）する．そして，サブゴールがこれ以上分解できなくなるまで繰り返す．最後に，ゴール指向分析よって導出した末端のゴール毎に難読化を適用する．本章で提案するアプローチは次の 5 つのステップで行う．

Step1. 攻撃者の能力モデルを定義する

Step2. 攻撃者のゴールを定める

Step3. ゴール指向分析を行う

Step4. ゴール指向分析によって導出された末端のゴール毎に適切な難読化を選択する

Step5. Step4. で選択した難読化を対象のプログラムに適用する

次節から，それぞれの工程について詳細に説明していく．

2.3 攻撃者の能力モデルの定義

セキュリティ機構にセキュリティゴールを設定する際に，現実的な攻撃者を想定し，その攻撃が実現可能（および，実現不可能）かどうかを示す必要がある．例えば，攻撃者はプログラムに使用されているセキュリティ機構の原理および，アルゴリズムを理解するために，実行可能なバイナリを持っている可能性がある．同様に，攻撃者はプログラムに対して逆アセンブラや逆コンパイラを用いて静的解析を試みるほか，デバッガのブレークポイント機能を用いて動的解析を行う可能性もある．

Monden らは，文献 [49] において，難読化手法を提案するにあたり，攻撃者の特徴的な知識や資産を次の 3 つの側面に従い，定義している [49]．(1) 使用されている保護のメカニズムへの理解度，(2) システムの観察に関するスキルレベル，(3) システムの制御に関するスキルレベル，である．これらの側面はソフトウェア保護メカニズムを評価するには有用に思える．本章では，(1) については，特定の保護手法についてのみしか想定していないため，対象外とする．ただし，(1) において，攻撃者の目標のシステムへの知識（理解度）を特徴づける必要がある．上記の議論を元に，本章では，仮想の攻撃者の能力モデルを (A) 知識，(B) 観察，(C) 制御の 3 つの側面から特徴づけを行う．以下に，非常に有能な攻撃者を想定したそれぞれの側面の能力の例を示す．

(A) システムの知識

攻撃者は，対象プログラムにおける原理，アルゴリズムと外部仕様などのすべての知識を持つ．

(B) システムの観測

攻撃者は攻撃対象のプログラム P のバイナリプログラム，逆アセンブルリスト，または，逆コンパイルによって得たソースコードと P を実行させる実行環境 M を所有する．攻撃者は，デバッガのブレークポイント機能によって P あるいは，実行環境 M の内部状態を観察することができる．たとえば，実行環境 M のメモリのスナップショット，P における入力値，および，出力値などである．また，攻撃者は P の実行履歴を観察することができる．すなわち，オペコード，オペランドの系列とそれらの値を観察することを意味する．

(C) システムの制御

攻撃者は，M に任意の入力を与え，そして，P を実行する．攻撃者は，P 内の命令を変更できるほか，オペランドの値や実行前，実行中の M のメモリイメージを思いのままに操作できる．

上記の攻撃者の能力モデルの説明においても，攻撃者は様々な攻撃手段を持っている．攻撃者は，P の逆アセンブルリストを調べ，P で使用されているアルゴリズムが実装されている箇所を探したり，P 実行中のスタック上にプッシュされた秘密情報の候補を見つけるためにスタックを観察したり [3]，また，入力の異なる複数の実行系列を収集し，それらを比較したりすることで，オペランド中にあらわれる秘密情報の候補を探索する可能性がある [73].

2.4 攻撃者のゴールを設定する

提案手法の第 2 段階は，攻撃者のゴールを同定することである．プログラム P について，攻撃者が P をリバースエンジニアする手段ごとに具体的なゴールを定義する．

ここに，DES が使われた典型的な暗号プログラム (図 5.2) の保護を仮定する．プログラムの入力は，暗号文の系列であり，例えば，それは暗号化されたディジタルメディアのコンテンツデータである．出力は，平文の系列であり，復号された音声および映像データである．DES は Feistel 構造をもつ対称鍵ブロック暗号である (図 5.2)．プログラムの内部では，暗号文が，L（上位ビット）と R（下位ビット）に分割される．そして，復号鍵 K からキースケジューラが復号のラウンド毎にラウンド鍵 k_n を生成する．その後，F 関数がそのラウンド鍵を入力と

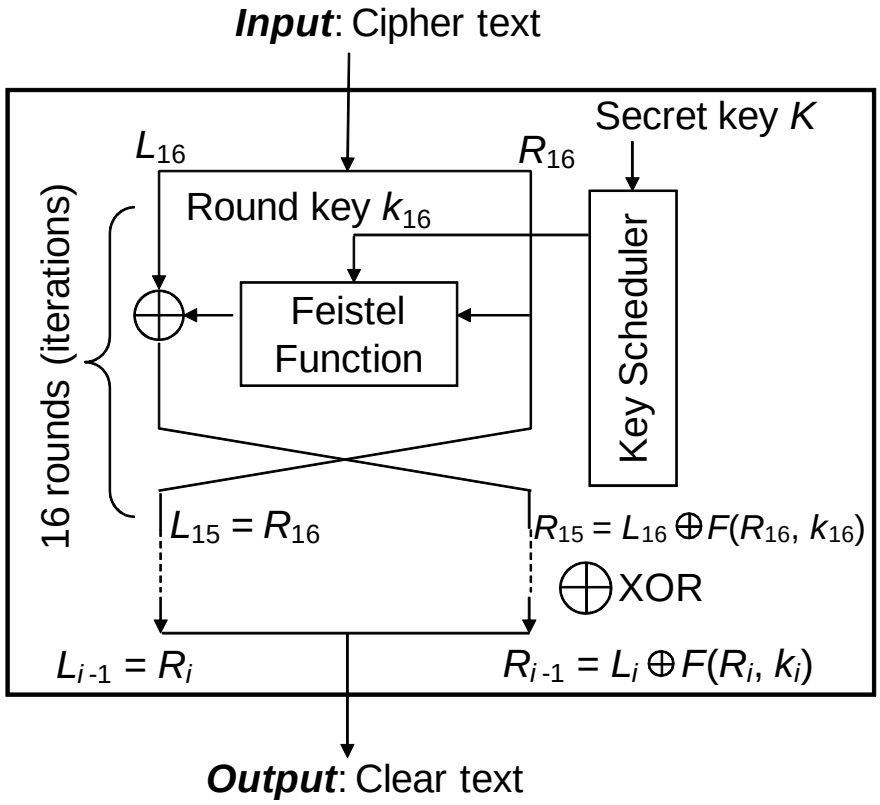
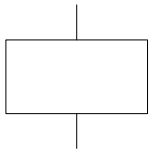
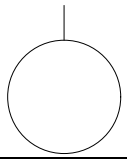
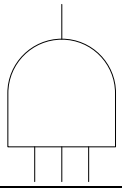
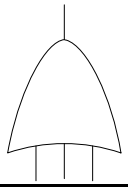
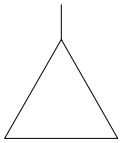
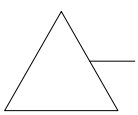


図 5.2 DES のブロック図

して，暗号文を復号する．この手順が 16 ラウンド繰り返され，データは復号される．

DES のプログラム中に攻撃者から保護すべき秘密情報は次のとおりである．(1) 復号鍵 K，(2) ラウンド鍵 $k_1, \ldots, k_{16}$ そして，(3)F 関数に含まれる非公開のデータ変換テーブルである．攻撃者のゴールはこれらの秘密情報をプログラムから抽出する事である．本章では，簡単化のために (2) ラウンド鍵を（攻撃者の）ゴールとして，提案する手法を説明していく．

表 5.1 ゴール木の表記記号

	**ルートゴール** 攻撃者が攻撃目標とするシステムの攻撃者の最終目標を表す.
	**中間ゴール** 親ノードのゴール（ルートゴールおよび, 中間ゴール）をより具体的に分解したサブゴール. 攻撃者はルートゴールを達成する前に中間ゴールを達成する必要がある.
	**末端ゴール** これ以上を分解できない具体的なゴール
	**AND ゲート** 上位のゴールを達成するために，下位のすべてのゴールを達成する必要がある事を示すゲート
	**OR ゲート** 上位のゴールを達成するために，少なくとも，1 つ以上の下位ゴールを達成する必要がある事を示すゲート
	**移行（入力）** 紙面上の都合，あるいは，ゴール木を分けたい時に用いる記号.
	**移行（出力）** 紙面上の都合，あるいは，ゴール木を分けたい時に用いる記号.

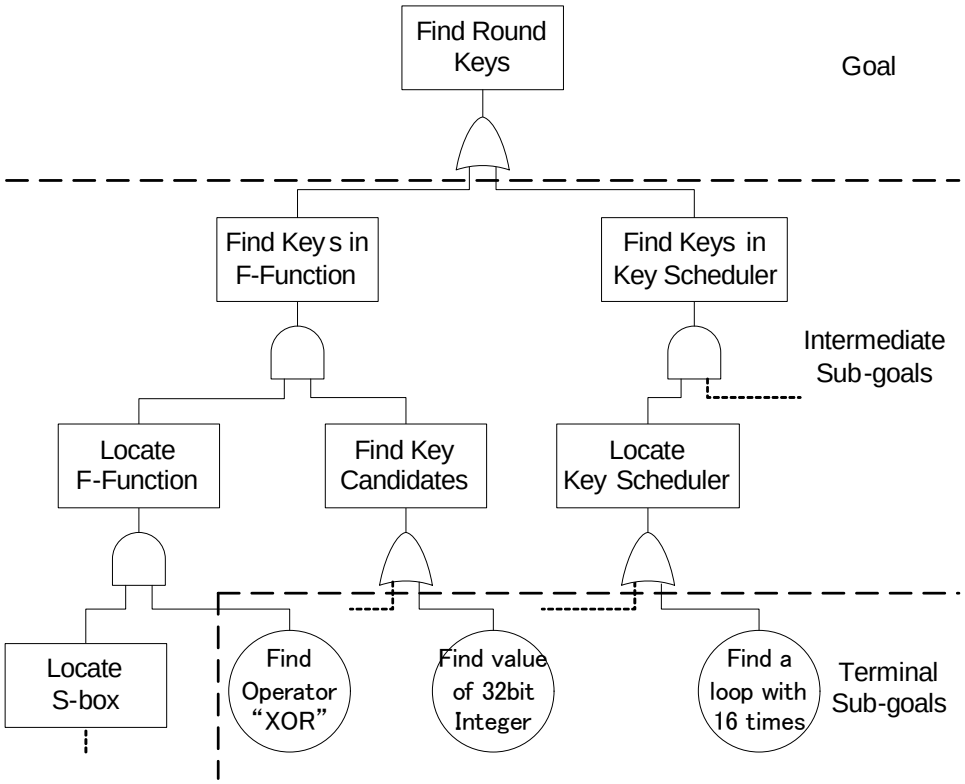


図 5.3 DES におけるゴール木

2.5 ゴール指向分析の実施

攻撃者はそのゴール（例えば，ラウンド鍵を探す）を達成するにあたって様々な手段（サブゴール）を持つ可能性がある．さらに，そのサブゴールにおいても同様に，より小さいサブ-サブゴールがあると思われる．そこで，提案フレームワークでは，ゴール木（AND-OR 木）はそれらのゴールを分解して形成する．

表 5.1 にゴール木に使用する記号とその意味を表す．これらの記号は主に故障系統図解析 (FTA) [68] で使われるものであり，障害が起こりうる事象をより具体化し，リスク分析に利用される．表 5.1 にゴール木に使用するシンボルを示す．ゴールには，3 つの種類のゴールがあり，(1) ルートゴール，(2) 中間ゴール，(3) 末端ゴール である．これらのゴールは，AND もしくは，OR のゲートによっても

う一方のゴールに接続される．もし，ゴールと下位のゴール（中間ゴールおよび末端ゴール）と AND ゲートにより接続されていたら，攻撃者が全ての下位ゴールを達成する必要があるという意味である．一方で，ゴールがもし OR ゲートによって接続されていたら，攻撃者は下位のゴールを少なくとも 1 つのゴールを達成する必要があることを意味する．もし，ゴール木が紙面に収まらないぐらい大きくなりすぎた時は，移行記号を用いて，これらゴール木を分割することが可能である．以下に基本的なゴール木の作成について説明する．

(1) ルートゴールをゴール木の一番上に置く．図 5.3 は，ゴール “Finding Round Keys” をルートゴールにしたゴール木の例である．

(2) 2.4 節で定義した攻撃者の能力モデルを元にルートゴールを中間ゴールへと分解を行う．図 5.3 では，ルートゴール “Find Round Keys” を，“Find Keys in F-Function” と “Find Keys in Key Scheduler” の 2 つの中間ゴールに分解した．何故なら，ラウンド鍵は F 関数と鍵生成部の 2ヶ所に現れるからである．

(3) (2) で分解したサブゴールと上位のゴールを AND または，OR ゲートで接続する．図 5.3 では，中間ゴール “Find Keys in F-Function” と “Find Keys in Key scheduler” は，上位のゴールと OR ゲートで結んだ．

(4) それぞれ分解されたゴールを，さらにサブゴールに分解できなくなるまで，ゴールを分解していく．その時，これ以上分解できないゴールを末端ゴールとする．全てのゴールについて分解できなくなるまで (2) . . . (4) の手順を繰り返す．

この手順は，トップダウンなアプローチであり，順方向に（ルートゴールから末端ゴールへ）ゴール木を生成している．しかし，ボトムアップなアプローチを同時に使用することで，ゴール木の作成の手助けになる．以下に，トップダウンなアプローチを補足するボトムアップアプローチの方法を説明する．

(1) 攻撃者の能力モデルによって定義した攻撃者が対象のプログラムから得られそうな手がかりを列挙する．ここで手がかりとは，ルートゴールを達成

に寄与する情報の断片である．例えば，XOR 演算は手がかりとなりえる．なぜなら，DES の暗号ルーチンに大量の XOR が現れるからである．

(2) (1) で，具体的な手がかりを元に，抽象化した手がかり，すなわち，中間ゴールを形成する．例えば，F 関数は 抽象的な手がかりになりうる．なぜなら，F 関数は大量の XOR 演算を含んでいるからである．ゴール “Locate F-Function” はトップダウンアプローチにより，中間ゴールとして分類されている (図 5.3) ので，“Finding Operator XOR” はそのサブゴールとして接続されている．

2.6 適切な難読化の選択

2.5 節で明確化した末端ゴール毎に，攻撃者の末端ゴール達成を阻止するために適切な難読化を選択する．図 5.4 に難読化手法の選定の例を示す．この例では，攻撃者からゴール “Find Operator “XOR”” の達成を阻止するために，演算の変換の難読化を採用する．同様に，ゴール “Find value of 32bit Integer” を阻止するために，データ型の難読化を採用する（32 ビットの整数を 4 つの変数に分割するなど）．同様に，ゴール “Find a loop with 16 times” を隠すために制御フローの難読化を採用する．

2.7 選択した難読化の適用

2.6 節で選択した全ての難読化手法を対象のプログラムに適用する．その結果，全ての末端ゴールを達成するのが困難になっている，つまり，全ての中間ゴールについても達成が困難になっている．同様にルートゴール “Find Round Keys” についても達成が困難になっている．

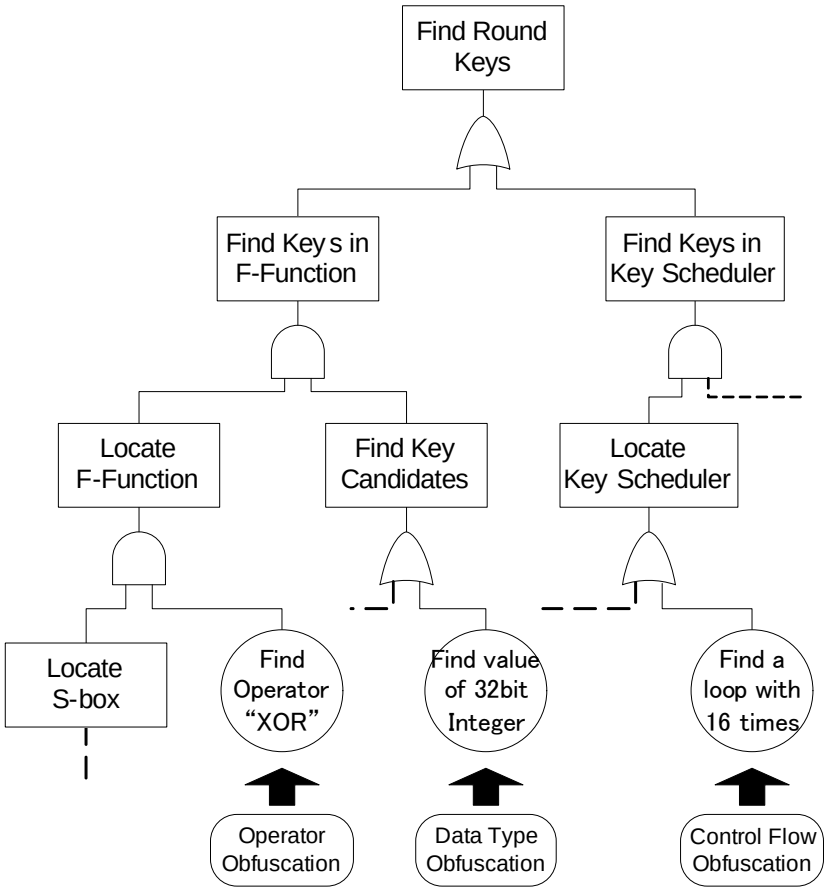


図 5.4 適切な難読化手法の選択

3. ケーススタディ

本節では，提案手法を実際のプログラムに適用し，どのように機能するかを説明するために，ケーススタディとして DRM システムで利用される暗号プログラムを例に難読化を行う．

3.1 対象プログラム

典型的な DRM システム例として cryptomeria cipher (C2) 暗号プログラムの保護を想定する．C2 暗号アルゴリズムの概要を図 5.5 に示す．さらに難読化適用対象の Java 言語で書かれたソースコードを図 5.8 に示す．このアルゴリズムは，CPPM および CPRM と呼ばれるコピー制御方式に利用されている [1]．図 5.5 において，C2 暗号は Feistel 構造の対称ブロック暗号であり，DES に類似している．図中の Feistel Function は，F 関数である．図 5.6 に C2 暗号の F 関数の詳細を示す．C2 暗号が DES と大きく異なる点は，C2 暗号には，加算と減算が存在する．本節の C2 暗号プログラムは例を単純にするため，鍵生成部を必要としない ECB (Electric Code Book) モードのソースコードである．

3.2 攻撃者のゴールと能力モデル

知識

C2 暗号アルゴリズムは公開されているので，攻撃者は独自で既存の DRM によって暗号化されたコンテンツを復号するプログラムを作成することができる．また，攻撃者のゴールはラウンド鍵を探すことである．C2 暗号の仕様書 [1] を読むことで，攻撃者は C2 暗号アルゴリズム (図 5.5) を理解でき，以下の知識を得ることができる．

- ラウンド鍵 k_i はそれぞれ鍵生成部ルーチンから生成されるか，または，直接プログラム P に定数として書かれる．前者において，デバイス鍵 K は P 内に存在するか，P の入力として与えられ，鍵生成部ルーチンに供給さ

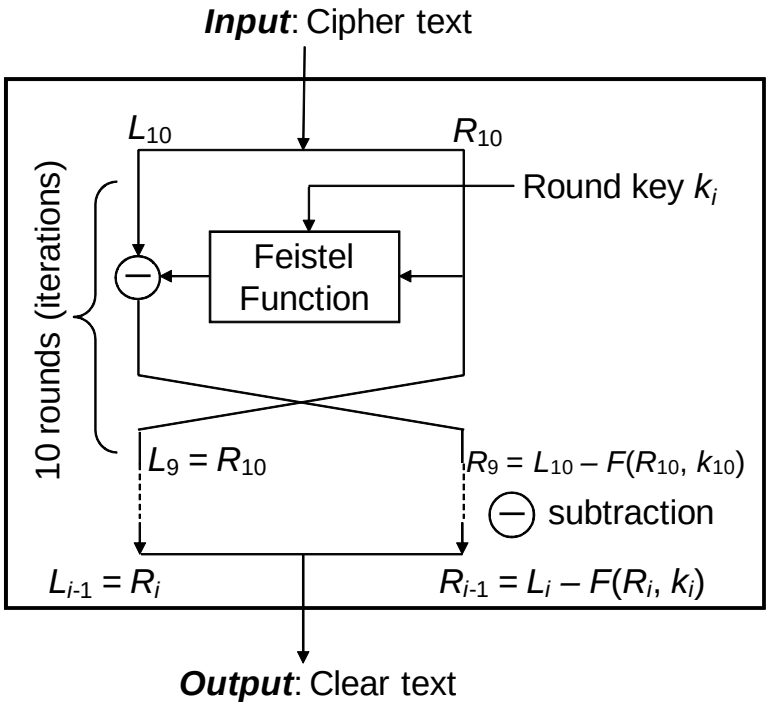


図 5.5 C2 暗号の概要

れる．後者において，デバイス鍵 K と鍵生成部ルーチンは P に存在しない可能性がある．本章における攻撃者のゴールは全てのラウンド鍵を探すことである．

- 10 回の繰り返しが存在し，それぞれにラウンド鍵 $k_1, \ldots, k_{10}$ が存在する．
- ラウンド鍵の鍵長は，32 ビットである．
- 入力のブロック長は 64 ビットであり，このブロックは上位 32 ビット L と下位 32 ビット R に分割される
- プログラム中の F 関数がある．L と R のどちらかが，それぞれのラウンドで F 関数の入力となる．

- L が F 関数の演算結果で減算される．さらに減算された値が，次のラウンドの R および，次のラウンドの F 関数の入力となる．

F 関数についても同様に，攻撃者は次の知識を持っていると予想される．

- L か R のいずれかが，ラウンド鍵に加算される．P 内に `add` 演算と 2 つの 32 ビットのオペランドがあることを示している．
- 加算の結果 X（ビット長 32 ビット）を 4 つの 8 ビットのブロック $x_1, \ldots, x_4$ に分割する．この分割は次のような記述により行われると考えられる．“x_1 = (X >>> 24) & 0xff, x_2 = (X >>> 16) & 0xff, x_3 = (X >>> 8) & 0xff, x_4 = X & 0xff.”
- 分割された最下位の 8 ビットブロック x_4 は，S-box によって，変換される．S-box とは，256 個の 8 ビットテーブルである． これは，256 個の要素を持つ配列が P 内にある事を示す．例えば，x_4 の変換は次のように表現できる．“`S-box_array[`x_4`]`”
- 残りの 3 つのブロック $x_1, \ldots, x_3$ は それぞれ 0xc9，0x2b，0x65 と XOR される．つまり，XOR の演算および，0xc9，0x2b，0x65 の定数が P に存在する事を意味する．その後，これらの 3 ブロックは，それぞれ，2 ビット，5 ビット，1 ビット の左ローテートが行われる．
- 4 つのブロックは結合されて 32 ビット の値に戻され，この値は，9 ビット，22 ビット の左ローテートが行われる．つまり，シフト演算，定数値 9, 22 が存在する事を意味する．また，4 つのブロックを結合する式は，“x_1 << 24 | x_2 << 16 | x_3 << 8 | x_4” によって行われると予想される．

観測と制御

攻撃者は逆アセンブラや逆コンパイラを静的解析に使用する．Sun Microsystems 社が提供している Java の逆アセンブラ（例えば，javap -c コマンド）や Meyer 氏が製作した jasmin 形式 [45] で出力される D-Java[46] でも同様に，逆アセン

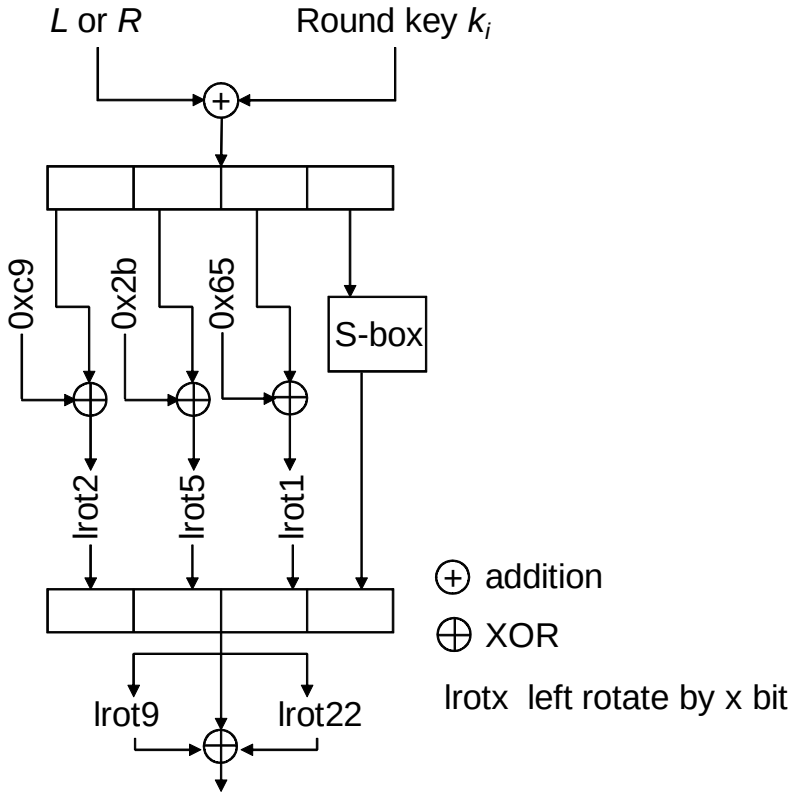


図 5.6 C2 暗号における F 関数のブロック図

ブルしたコードを再度アセンブリし，jasmin アセンブラ [47] や Soot[58] によってクラスファイルに戻すことができる．また，各種の Java 逆コンパイラが利用可能だが，ほとんどの場合，不完全なソースコードが生成されてしまう．そこで，攻撃者はデバッガを使用すると考えられる．Java のデバッガとしては，Sun Microsystem 社の jdb, Hex-Rays 社の IDA Pro[30] あるいは，Bil Lewis 氏の Ominiscient Debugger[37] などがある．

ゴール指向分析

C2 暗号プログラム (図 5.8) において，攻撃者のルートゴールを “find round keys” に想定し，これを難読化する必要がある．このルートゴールに基づいて，ゴール指向分析を導入し，ゴール木の作成を行う (図 5.7)．以降，トップダウンアプローチでゴール木の作成の例を示す．

ルートゴール “Find Round Keys” を達成する手段は 2 つある．1 つは，静的解析によって “Find Key Candidates in constants” であり，もう 1 つは動的解析によって “Find Key Candidates in variables” である．これら 2 つの手段を図 5.7 では，ルートゴールのサブゴールとして設定する．

サブゴール “Find Key Candidates in constants” を達成するため，ラウンド鍵の候補が P のどこに特徴的な定数として現れるかに注目した．上記の特徴は，ゴールを達成できる可能性のある手がかりとして考える事ができる．例えば，3.2 節で述べたように，各ラウンド鍵の鍵長は 32 ビットであることなどである．実際のプログラムでも同様に，鍵の値が配列 `sk[]` によって割り当てられ，32 ビットの定数が複数入っている（図中の `int sk[]` = 0x789ac6ee, 0x79bc3398, ⋯ ）．これら 32 ビットの定数値は，プログラムを逆アセンブルすることで発見されるであろうが，逆アセンブリコード中の全ての 32 ビットの定数がラウンド鍵ではない．したがって，“Find 32bit Integers” は，“Find Key Candidates in constants” の下位のゴールであるといえる．さらに，このゴールは，これ以上分解できないために末端ゴールだと考える．

次にゴール “Find Key Candidates in variables” の別の手段について考える．変数として，潜在的にラウンド鍵の値を処理している場所が 2 つある．(1) F 関数への入力と (2) F 関数内での鍵操作である．実際のプログラム (図 5.8) においても，ラウンド鍵の値は配列 `sk[]` に保持され，F 関数内に引き継がれる．F 関数内部でも同様に，ラウンド鍵の値は復号に使われる．これらの手段は，攻撃者の知識と能力 (3.2 節) に基づいて発見されうる．したがって，2 つのゴール “Inspect inputs of F-Function” と “Inspect key manipulation in F-Function” を “Find Key Candidates in variables” のサブゴールに設定した．

2 つのゴール “Inspect inputs of F-Function” と “Inspect key manipulation in

F-Function” に関して，攻撃者は F 関数を調べる必要がある．したがって，“Find Feistel Function” を下位ゴールに設定した (図 5.7)．ゴール “Find Feistel function” の達成に寄与する可能性のある 2 種類の手がかりについて考える．(1) F 関数の呼び出しに関係する手がかりと (2) F 関数自身が持つ特徴による手がかり である．

F 関数の呼び出しの手がかりについて，ゴール “Identify 10 times loop” と “Identify SUB operator” が大きな手がかりとなる．F 関数は C2 暗号アルゴリズムから 10 回呼び出され，減算は F 関数呼出し後に行われるためである．F 関数自身の手がかりについては， F 関数内の特徴的な項目に注目した．それぞれ，(1) 特徴的な値，(2) 加算，(3)S-box，(4)32 ビットローテート演算，(5)8 ビットローテート演算，(6) 結合関数の 6 つである．

攻撃者はこれらの手がかりのうちの 1 つを実際のプログラム領域から見つけた場合，この領域を F 関数と考える可能性がある．さらに，攻撃者は，同じ箇所，または，その付近からその他の手がかりを探そうと試み，F 関数であるか確証を得ようとする．そのため，これらすべての手がかりをゴール木に追加すべきであり，これらを難読化する必要がある．以降のステップでこれらの難読化を行う．

最後までゴールを分解したものが図 5.7 である．それぞれのサブゴールについて，サブゴールを達成しようとする攻撃者の行動，および，任意の手がかりがサブゴールの達成に寄与するかを考察し，さらなるゴールの分解の可能性ついても調査した．もし，ゴールがこれ以上分解できない場合，そのゴールは末端ゴールとして設定した．その結果，合計 27 個のサブゴールができ，そのうち，12 個が中間ゴール，15 個が末端ゴールであった (図 5.7)．

適切な難読化の選択

ここでは，ゴール木中の 4 つのサブゴール (図 5.7 中 A, B, C, D) について難読化を考える．

特徴的な値の難読化　この節では，サブゴール “Identify Distinctive values 0x65, 0x2b, 0xc9” (図 5.7 中の A) 達成の攻撃者の行動の抑止を考える．このサブゴールは，上位ゴール “Find Feistel Function” を達成に寄与するゴールである．なぜなら，

定数 0x65, 0x2b, 0xc9 はそれぞれ，C2 暗号アルゴリズムの F 関数 (図 5.6) 内に現れると予想される．図 5.8 において，これらの 3 つの定数は `F(int data, int key)` 中に現れる．このメソッド中に 0x65 は以下の記述で現れる．

```
u = (byte)(v[0] ^ 0x65);
```

この 0x65 を隠すために，0x65 を 2 つの値 0x21 と 0x44 に分割する（0x65 = 0x21 XOR 0x44 を満たす）．そして，この 2 つの値を用いることで，上記の記述は以下の様に置き換えられる．

```
u = (byte)(v[0] ^ 0x21);
u = (byte)(u ^ 0x44);
```

同様に，他の 2 つの定数 0x2b と 0xc9 についても隠蔽する．この 2 つの定数についても “0x2b = 0x28 XOR 0x03” と “0xc9 = 0x41 XOR 0x88” という関係を利用する．それにより難読化した結果が図 5.9 の A の部分である．

10 回のループの難読化　次に，サブゴール “Identify 10 times loop” の達成の抑止について考える．このサブゴールは上位ゴール “Find Feistel Function” の達成に寄与する．なぜなら，F 関数は 10 回 C2 暗号アルゴリズム内で呼ばれる (図 5.5)．ここでは，単純にループをインライン展開し，10 回のループを除去した (図 5.9 の B)．

連結関数の難読化　次に，連結関数を難読化を行う (図 5.7 中の C)．連結関数は，4 つの 8 ビットの値を 32 ビットに結合する．最も単純な実装方法は，`t = v[3] << 24 | v[2] << 16 | v[1] << 8 | v[0]` （v は配列，4 つの 8 ビットの値を保持しており，t に演算結果のが格納される）である．攻撃者は連結関数から，次のような手がかりを得ると考えられる．例えば，24, 16, 8 の定数，左シフト演算 (<<)，そして，OR 演算 (|) である．実際に，図 5.8 において，連結関数はを以下の様に実装されている．

```
t = (int)v[3] << 24 | (int)v[2] << 16 | (int)v[1] << 8 | (int)v[0];
```

ビットシフト演算 (<<) を除去するために，その代替として乗算を用いる．左ビット n シフトは，乗算 2^n に置き換えられる．例えば，`v[3] << 24` は，`v[3] * 16777216` に置き換えられる．この置き換えによって，特徴的な値 24 は除去できる（$16777216 = 2^{24}$ と代替される）．OR 演算 (|) は，ド・モルガンの法則を用いて NOT (~) と AND (&) 演算に置き換えた．結果として，以下のようになった．

```
t = ~(((~((int)v[3] * 16777216) & ~((int)v[2] * 65536)
    & ~((int)v[1] * 256)) & ~((int)v[0]));
```

32 ビット整数型の難読化　次に，32 ビットの整数の難読化を行う．その理由は，ラウンド鍵は 32 ビットの整数型であるからである (図 5.7 の D)．攻撃者は逆アセンブルより得たリスト，または，スタックの中からラウンド鍵を探すため，あらゆる 32 ビット値の出現を監視する可能性がある [3]．この攻撃からラウンド鍵を保護するため，鍵の値を変換すべきである．図 5.9 の D は，線形写像によって準同型変換 [57] を `sk[]` に適用したものである．この方法はそれぞれの値を 1 次関数で変換する（ここでは，変換式 $f(x) = 4x + 3$ を用いた）．この変換でラウンド鍵 `0x789ac6ee` は，`0x3ffb81617L` にエンコードされる．エンコードされた鍵は，F 関数の以下の記述で使用される．

```
t = data + key;
```

エンコードされた鍵で計算するには，`t = data + key` の記述を `t = data * 4 + key` に置き換えればよい．それによって t は，エンコードされたままの値で保持できる．元来の t の値が必要になる時は，変換式の逆関数 $f^{-1}(x) = (x-3)/4$ を計算する．これにより，t は復号され，元の値になる．

ここで重要な事は，計算の過程において元に鍵の値 `0x789ac6ee` が 定数，変数，スタックのいずれかに現れることである．一次変換を用いる代わりに，中国剰余定理，bit exploded encoding [18], 秘密共有準同形変換 [6][56], 変数の結合 [19] やエラー訂正符号 [42] などを利用して，鍵の値を隠蔽することも考慮に入れている．

準同型変換あるいは，その他の手法によって変換された鍵は，その後，変換した鍵の値を加工（例えば，いくつかの値を分割する）などして，攻撃者に鍵の候補として発見させないように実装を行うべきである．

4. 難読化の隠ぺい

全ての難読化の適用が完了した後，すなわち，ゴール木の末端ゴールからルートゴールへの全ての経路を難読化によって阻止ができたら，秘密情報が発見できる手がかりが存在するかどうか，難読化を適用したプログラムを注意深く確認する必要がある．難読化手法そのものが特徴的なので，攻撃者に難読化が使用されているかを悟られないために行う必要がある．例えば，ラウンド鍵を隠蔽するために剰余定理を用いたエンコード [18] を採用した場合，多くの剰余演算が難読化されたプログラム内に現れる．この場合，剰余演算をより単純な剰余ルーチンに書き換えるなどして，これらをさらに難読化する必要がある．

5. 評価

本節では，提案難読化フレームワークの有効性について評価をする．評価では，既存の難読化ツールにより自動難読化を行った場合と，難読化フレームワークにより難読化すべき場所を特定して人手により難読化を行った場合の比較を行う．

5.1 実験方法

評価対象は，ケーススタディで使用した C2 暗号のソースコード (図 5.8) である．比較対象となる自動難読化ツールとしては，SandMark[15] を用いる．SandMark は，プログラムへの電子透かしの挿入と難読化を行うツールであり，39 種類の難読化手法が実装されている．今日知られている難読化手法のうち，自動化できるものについては，ほぼ網羅的に実装されている（39 種類の難読化手法の詳細については付録を参照されたい.）

実験では，まず，評価対象のソースコードをコンパイルし，jar ファイルを作成する．作成した jar ファイルを入力として， SandMark を用いて 39 種類の難読化手法を 1 つずつ適用する．それぞれの難読化手法が適用された jar ファイルを `javap` コマンドの `-c` オプションでクラスファイルを逆アセンブルして静的解析するとともに，デバッガおよび実行系列出力ツール AddTracer[64] を用いて動

的解析を行い，3 つの秘密情報（ラウンド鍵の値，F 関数を発見する手がかりとなる定数 0x65，および，演算<<24）が観測できるか確認をした．

一方で，提案フレームワークにより難読化すべき場所を特定し，人手による難読化を適用したソースコードとして，ケーススタディで難読化した図 5.9 のソースコードを用いた．こちらもソースコードをコンパイルし，jar ファイルを作成後，静的解析，動的解析をそれぞれ行い，秘密情報が観測できるか確認した．

5.2 結果

評価実験の結果を表 5.2 に示す．39 種類の難読化手法のうち，Class Encrypter 以外は静的解析によりラウンド鍵，定数 0x65，演算<<24 のいずれの秘密情報も観測できることが分かった．また，Class Encrypter は，動的解析によりいずれの秘密情報も観測できることが分かった．

さらに，Class Encrypter については，jar ファイルからクラスファイルを抽出した結果，クラスローダの機能を持つクラスファイルは暗号化されておらず，他の（暗号化された）クラスファイルを復号する手続きを含んでいることを確認した．このことから，復号後のクラスファイルの情報を標準出力やファイルに出力するようにクラスローダを書き換えることで，暗号化されていたクラスファイルの静的解析が可能となることが分かった．

一方，提案方法では，ゴール木に基づいて難読化を行っているため当然ではあるが，静的解析と動的解析のいずれにおいても 3 つの秘密情報は観測できなかった．

5.3 考察

本実験の結果から，自動化できる難読化手法を適用したとしても，秘密情報（ラウンド鍵，定数 0x65，演算<<24）を隠蔽できるとは限らないことが分かった．ラウンド鍵や 0x65 などのデータは，原理的には（静的に）容易に隠蔽できる（例えば，0x65 を 0x60 と 0x05 の足し算に置き換えるなど）が，このような置き換えはあまりにも単純であるため，難読化手法としてはそもそも提案されておらず，SandMark にも実装されていなかった．また，仮にこのような難読化手法が実装さ

れていたとしても，動的解析から隠蔽することはできない．動的解析からデータを隠蔽するためには，本論文のケーススタディで行ったように，準同型写像を用いてデータを変換し，変換したまま演算を行うといった処理が必要である．もしくは，ビットシフト命令を乗算に置き換えたように，演算のやり方そのものを変更する必要がある．ただし，このような難読化は，自動化が必ずしも容易でない．

以上のことから，秘密情報を隠蔽するためには，自動化できる難読化手法に頼るだけでは不十分であり，自動化できない（手作業による）難読化を行うことも不可欠である．そのためには，プログラム中のどの箇所をどのように難読化すべきかを知る必要があり，本論文で提案するゴール指向分析のフレームワークでは有効であることが分かった．

6. まとめ

本章では，ゴール指向分析により系統的に既存の難読化手法を適用し，秘密情報を保護する手法を提案した．本章では，どのように本フレームワークを暗号プログラムに適用すればよいかを考察するため，ケーススタディとして C2 暗号を用いて適用例を示した．また，ケーススタディで用いたプログラムを既存の難読化ツールにより自動難読化を行った場合と，難読化フレームワークにより難読化すべき場所を特定して人手により難読化を行った場合の比較実験を行った結果，自動難読化では手がかりが隠蔽できておらず，提案フレームワークによる人手による難読化が必要なことが分かった．

以上のケーススタディや比較実験により，提案フレームワークにより次の効果が得られることが分かった．

- 対象プログラムに対して攻撃者のゴールを設定し，ゴール分解を行うことによって，難読化すべき箇所が明らかとなる．
- 難読化すべき情報の種類（定数，演算，ループなど）に基づいて，どのような難読化を適用すればよいのかが明らかとなる．

- 比較実験を通して，従来の自動難読化と比べて，秘密情報をより的確に隠蔽できる．

提案フレームワークの限界としては，あらかじめ想定できない（思いもよらないような）攻撃手段または，ゴールについては，対策を講じることができない点にある．ただし，従来は，どのような難読化をどこに適用すべきか，また，それによってどの程度の効果が得られる，といったことの議論すらできなかったため，想定できる攻撃を整理し対策できるようになったことは大きな進歩であると考える．現状では，想定できない攻撃手段に対しては，自動難読化手法をできるだけ数多く適用するといった対策が考えられる．

表 5.2 SandMark による難読化と提案方法による難読化における手がかりの観測の有無

難読化手法	ラウンド鍵	定数 0x65	演算 <<24
Array Folder	○	○	○
Array Splitter	○	○	○
BLOAT	○	○	○
Block Marker	○	○	○
Bludgeon Signatures	○	○	○
Boolean Splitter	○	○	○
Branch Inverter	○	○	○
Buggy Code	○	○	○
Class Encrypter	△	△	△
Class Splitter	○	○	○
Constant Pool Reorderer	○	○	○
DuplicateRegisters	○	○	○
Dynamic Inliner	○	○	○
False Refactor	○	○	○
Field Assinment	○	○	○
Inliner	○	○	○
Insert Opaque Predicates	○	○	○
Integer Array Spilitter	○	○	○
Interleave Methods	○	○	○
Irreducibility	○	○	○
Merge LocalIntegers	○	○	○
Method Merger	○	○	○
Objectify	○	○	○
Opaque Branch Insertion	○	○	○
Overload Names	○	○	○
Param Alias	○	○	○
Promote Primitive Registers	○	○	○
Promote Primitive Types	○	○	○
Public Fields	○	○	○
Random DeadCode	○	○	○
Rename Registers	○	○	○
Reorder Instructions	○	○	○
Reorder Parameters	○	○	○
Simple Opaque Predicates	○	○	○
Split Classes	○	○	○
Static Method Bodies	○	○	○
String Encoder	○	○	○
Transparent Branch Insertion	○	○	○
Variable Reassigner	○	○	○
提案手法	×	×	×

○は静的解析で観測可能
△は動的解析で観測可能
×は観測不可を表す．

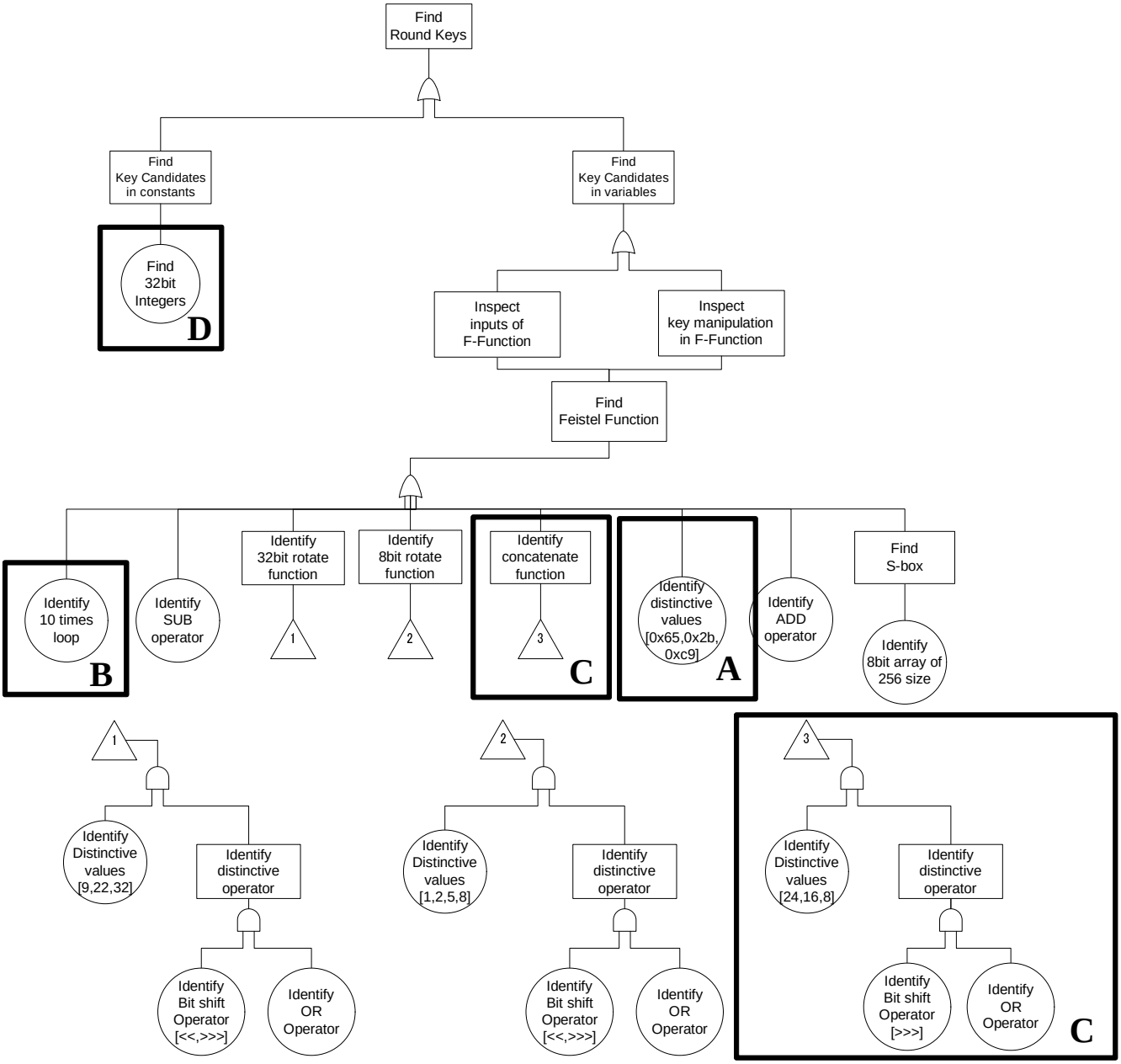


図 5.7 C2 暗号におけるゴール木

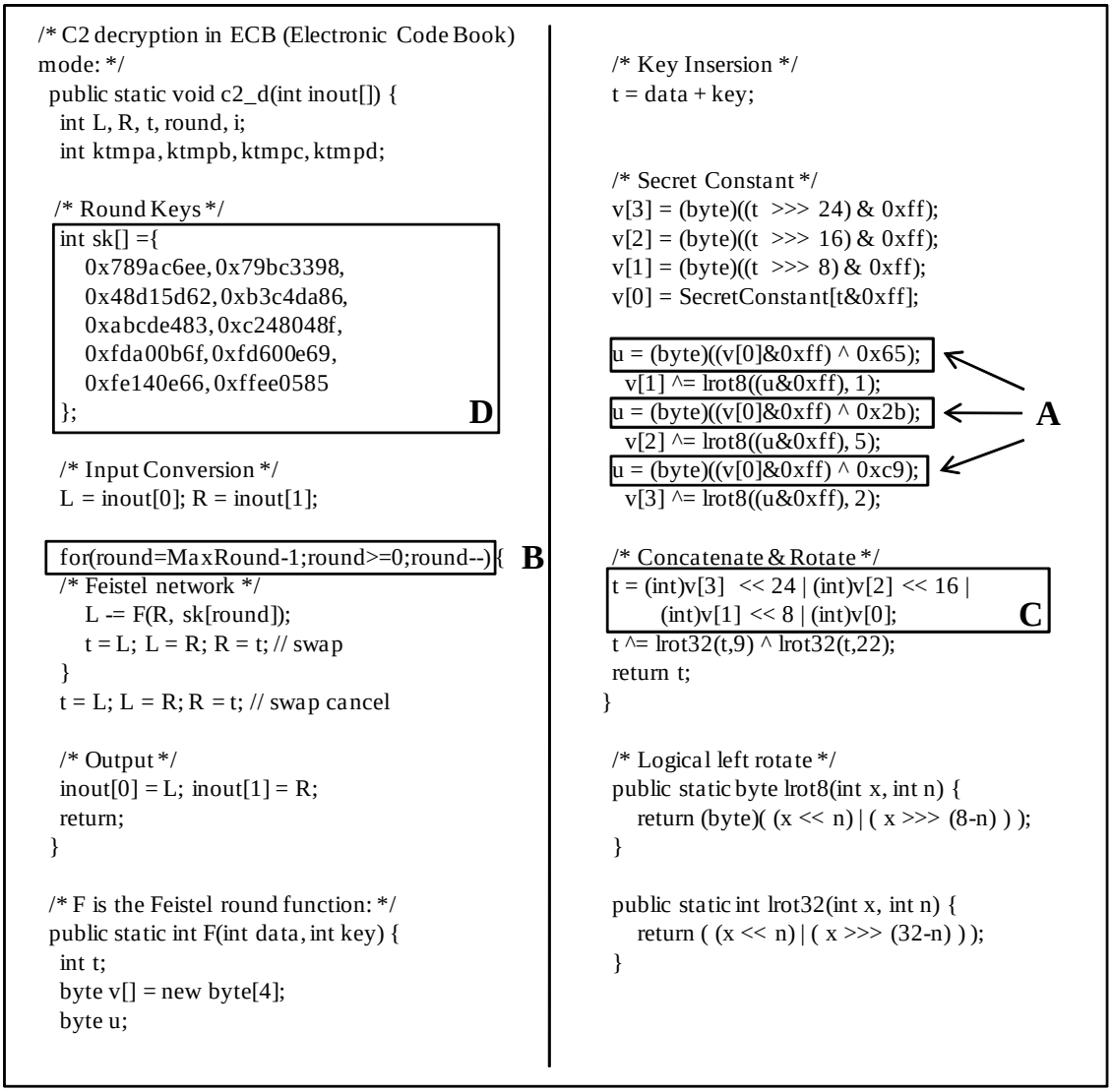


図 5.8 難読化適用前の C2 暗号を実装したソースコード

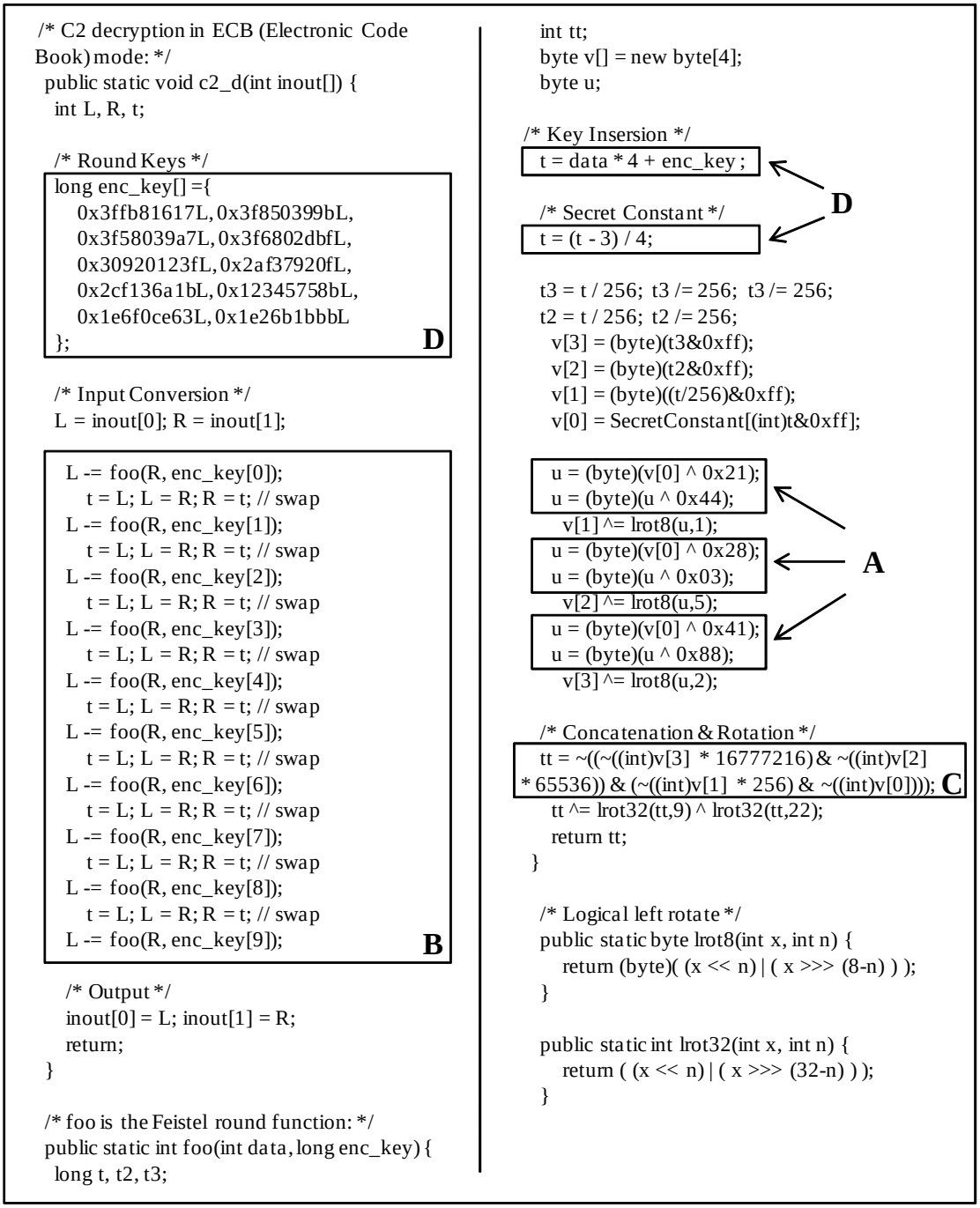


図 5.9 難読化適用後の C2 暗号を実装したソースコード

第6章　難読化フレームワークの拡張

1. はじめに

前章で提案した難読化フレームワークでは，ゴール木を構築していくことによって攻撃者の行動を漏れなく列挙し，それらを妨げるように難読化を行う．ただし，ゴール木の作成方法は必ずしも明確となっておらず，攻撃を全て網羅するようなゴール木の作成は，ゴール木の作成者の能力に依存している．より系統的なゴール木の作成方法が求められる．

難読化フレームワークにおいて，ゴール木の各ノードを構成する「攻撃者の行動」に着目すると，全ての行動は，プログラム中から秘密情報もしくは秘密情報の発見の手がかりとなる情報を探す，といった行動であり，ゴール木は事実上手がかりの連鎖を表す木となっている．このことは，プログラム中の手がかりを網羅的に列挙できれば，網羅的なゴール木の作成が可能となることを示している．

そこで，本章では，プログラム中の攻撃の手がかりを網羅的に列挙し，それらを難読化によって隠蔽するための拡張難読化フレームワークを提案する．本フレームワークでは，秘密情報とその手がかりとの関係，及び，手がかり間の関係を，アルゴリズム，ソースコード，バイナリの3つの抽象レベルに分けて記述できる．各レベルにおいて難読化により手がかりを隠蔽することで，秘密情報の発見を困難にすることができる．

2. 拡張難読化フレームワーク

2.1 攻撃の手がかり間の関係の記述

攻撃の手がかりとは，秘密情報と関係を持ち，秘密情報の発見に役立つ情報のことである．手がかりの発見に役立つ情報もまた手がかりである．図 6.1 に示すように，秘密情報と手がかりの関係，および，手がかり間の関係は，(1) 部分－全体，(2) 抽象－具体，(3) その他の 3 つに分類され，Unified Modeling Language (UML) のクラス図の記法により表現できる．

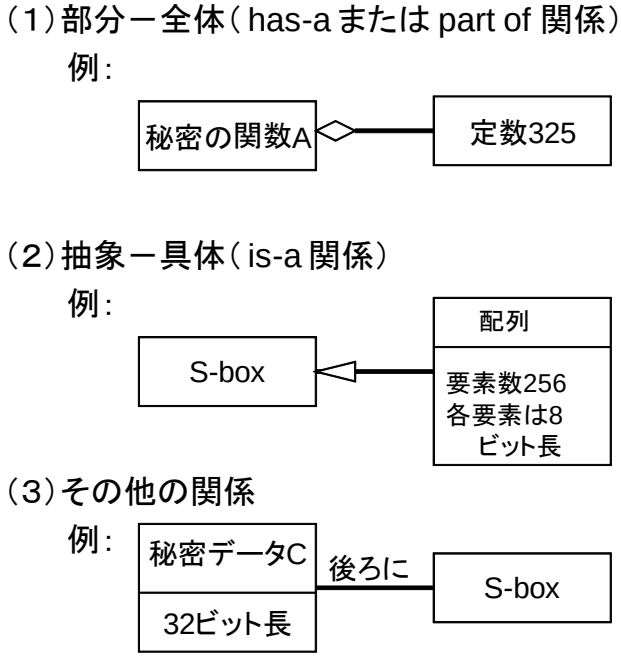


図 6.1 秘密情報と手がかりの関係

(1) の部分－全体関係では，手がかりは秘密情報の構成要素であり，図 6.1 に例示するように，秘密の関数 A の構成要素に定数 325 が含まれる場合，定数 325 を発見することが関数 A の発見につながる．(2) の抽象－具体関係では，手がかりは秘密情報を具体化したものである．例えば，S-box が配列（要素数 256，各要素

が 8 ビット長）により実現されている場合，配列を発見することが S-box の発見につながる．また，(3) のその他の関係としては，ある手がかりが別の手がかりを「呼び出す」という関係や，ある手がかりが別の手がかりの前（または後ろ）にあるという位置関係など，関連自体が別の手がかりを発見する手がかりとなる場合に記述する．図 6.1 の例では，例えば，秘密データ C の後ろに（すなわち，秘密データ C が出現した後の処理で）S-box が存在する場合，S-box の発見が秘密データ C の発見につながる．(1)，(2)，(3) のいずれの関係においても，難読化によって手がかり（もしくは手がかり間の関係）を隠蔽することで，秘密情報や他の手がかりの発見をより困難にできる．

秘密情報および手がかりは，それぞれ 0 個以上の属性を持つ．ここでいう属性とは，秘密情報および手がかりの特徴を現す情報であり，例えば，図 6.1(3) に示すように，秘密データ C が 32 ビット長である場合,「秘密データ C」の属性は「32 ビット長」となる． 32 ビット長の秘密データ C を 8 ビットずつ保持するなどの方法により，属性を隠蔽することで，秘密情報や手がかりの発見防止につながる．同様に，S-box を実現する配列の属性は,「要素数 256」および「各要素は 8 ビット長」となる．

2.2 抽象度による手がかりの整理

前節の抽象－具体関係でも例示したように，それぞれの手がかりは抽象度が異なる場合がある．本論文では，アルゴリズムレベル，ソースコードレベル，機械語レベルの 3 つの抽象度に分けて手がかりを整理する．図 6.1 に出てきた秘密情報および手がかりを抽象度によって整理した例が図 6.2 である．このように，抽象度によって手がかりを整理することで，手がかり間の関係が分かりやすくなるともに，異なる抽象レベルの手がかりを網羅的に列挙しやすくなる．

図 6.2 の S-box に着目すると，S-box は暗号アルゴリズム上の概念であるため，アルゴリズムレベルの手がかりである．これを実装した配列（要素数 256，各要素が 8 ビット長）はソースコードレベルの手がかりとなる．また，この配列の実装に対応する機械語プログラム上の特徴「`var_xx = xxx` が 256 個続く，`STR R3,[R11,#var_xx] MOV R3,#0` が 256 個続く」が機械語レベルの手がかりと

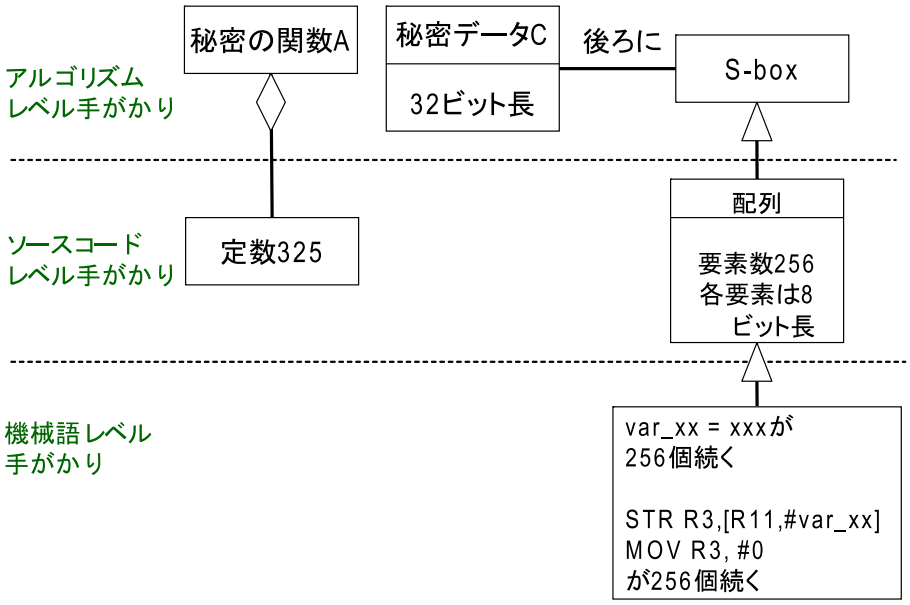


図 6.2 手がかりの抽象度

なる．

3. ケーススタディ

ケーススタディとして，第 5 章の第 3 節の C2 暗号プログラムにおいてラウンド鍵を隠蔽する場合の例を図 6.3 示す．まず，秘密データであるラウンド鍵は 32 ビット長であるため,「ラウンド鍵」の属性に「32 ビット長」と記載している．ラウンド鍵は，Round 関数および Feistel 関数で使われるため，それらから「use」という関係の線を引いている．また，Round 関数は Feistel 関数を呼び出すため，これらの間に「call」という関係がある．これらの手がかりはいずれもアルゴリズムレベルの手がかりである．

次に，Round 関数に着目する．Round 関数は「10 回のループ」,「Swap」,「減算 (-)」といった手続きを含み，これらが Round 関数を発見する手がかりとなる．

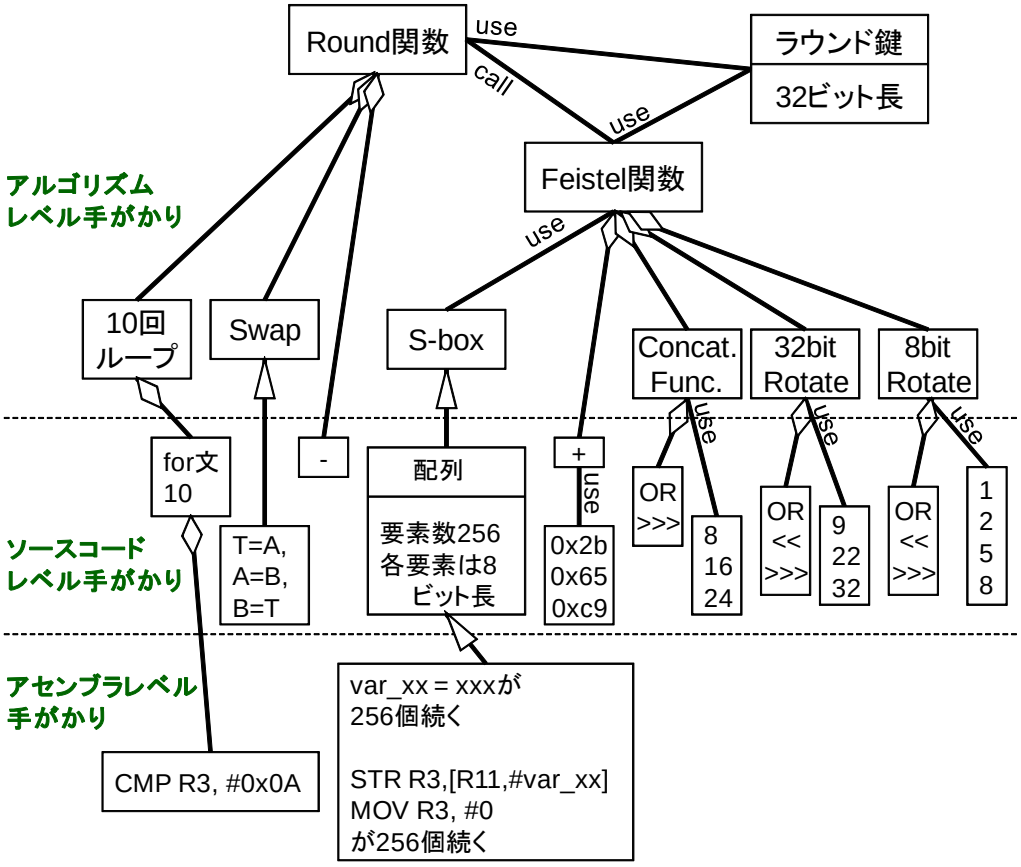


図 6.3 C2 暗号プログラムにおける手がかり間の関係図

そこで，これらの手続きを手がかりとして図中に記す．これらはいずれも Round 関数の構成要素であるため，部分-全体 (has-a) の関係の線を引いている．また，図中では，これらの手がかりのうち「10 回ループ」と「Swap」はアルゴリズムレベルの手がかりであり，減算はソースコードレベルとして区別している．さらに,「10 回ループ」は，ソースコードレベルでは for 文として実現される．また，これをコンパイルして得られる機械語レベルでは,「CMP R3,#0x0A」などと表される（ARM プロセッサの場合）．これらの関係を図示することで，攻撃者が 10 回ループを手がかりに Round 関数を発見しようとする場合，プログラムを逆アセンブルしてCMP R3,#0x0A となっている箇所を探索する可能性があることが分かる．

以降，同様に，各手がかりの構成要素を列挙したり，各手がかりを具体化した要素を列挙していくことで，隠蔽すべき手がかりを網羅的にあぶり出すことが可能となる．図 6.3 では，前章の例 (図 5.7) と比較して，手がかり間の関係がより分かりやすくなっている．また，前章の例では抜けていた「Swap」という手がかりを列挙できている．さらに，図 6.3 では，手がかり間の関係が具体的に記されているため，難読化によって手がかり自体を隠蔽するのみならず，手がかり間の「関係」を隠蔽することも可能となっている．例えば，Round 関数は Feistel 関数を call するという関係にあるが，これを隠蔽するために，Round 関数が Feistel 関数を直接 call しないように難読化することが可能である．

4. 提案フレームワークの適用範囲

本論文で提案した難読化フレームワーク，および，その拡張フレームワークは，特にプログラムの静的解析攻撃に対して有用である．静的解析では，プログラム中から特徴的な手がかりを探し，そこからプログラムを読み進めるが，それらは部分-全体，抽象-具体，呼び出し，位置（前後）といった関係として表現することが可能なためである．

一方，動的解析では，プログラムの挙動に関する手がかりをまず探し，そこからプログラムを読み進めることとなる．例えば，パスワード入力ウィンドウのポッ

プアップ，サーバとの通信の発生，認証失敗を示すウィンドウのポップアップなどである．これらの手がかりは，難読化によって隠蔽することがそもそもできないため，提案フレームワークの対象外となる．これら動的な手がかりについては，4 章で示した段階的プロテクションにおいて，外部仕様のプロテクション，内部仕様のプロテクションにおいて対策を施すことが必要となる．

第7章　おわりに

本論文では，ソフトウェアシステムの系統的な保護を目的として，まず，第2章では，エンドユーザによるソフトウェアシステムに対する攻撃方法を，第3章では，保護方法について整理した．その結果に基づいて，第4章では，ソフトウェアの各開発工程において段階的にプロテクション技術を適用するためのガイドライン（段階的プロテクション）を提案した．

次に，第5章で段階的プロテクションの実施手順において重要となるソフトウェア難読化に着目し，ソフトウェア難読化手法を適材適所に適用するための枠組み（難読化フレームワーク）を提案した．提案フレームワークでは，攻撃者の攻撃におけるゴールを定義するとともに，ゴール達成に必要なサブゴールを，ゴール分解により求めていくことでゴール木を生成する．そして，得られたゴール木の全ての末端のサブゴールについて，その達成を妨げるのに必要な難読化手法を選定する．また，ケーススタディとして，秘密を含む典型的な Digital Rights Management (DRM) ソフトウェアの1つである cryptomeria cipher (C2) 暗号プログラムにおいて，復号鍵を隠蔽するためのゴール木を生成する事例を通して，多数の難読化手法を適材適所に適用できることを示した．

第5章の難読化フレームワークにおいて，ゴール木の各ノードを構成する攻撃者の行動に着目すると，全ての行動は，プログラム中から秘密情報，もしくは秘密情報の発見の手がかりとなる情報を探す，といった行動であり，ゴール木は事実上手がかりの連鎖を表す木となっていることがわかった．そこで，第6章では，プログラム中の攻撃の手がかりを網羅的に列挙し，それらを難読化によって隠蔽するため，フレームワークの拡張を行った．これにより，秘密情報とその手がかりとの関係，及び，手がかり間の関係を，アルゴリズム，ソースコード，バイナリの3つの抽象レベルに分けて記述し，各レベルにおいて難読化により手がかり

を隠蔽することで，秘密情報の発見を困難にできると考えられる．

謝辞

本研究を進めるに当たり，研究の方法などに関する多くのアドバイスと共に丁寧なご指導を賜りました，奈良先端科学技術大学院大学 情報科学研究科 ソフトウェア工学講座 松本 健一 教授に深謝致します.

本研究を進めるに当たり，的確で貴重なご助言を頂きました 奈良先端科学技術大学院大学 情報科学研究科 情報基礎学講座 関 浩之 教授に深謝致します.

本研究を進めるに当たり，多くの技術的なアドバイスに加え，細部にわたる熱心なご指導を頂きました，奈良先端科学技術大学院大学 情報科学研究科 ソフトウェア工学講座 門田 暁人 准教授に深謝致します.

本研究を進めるに当たり，多くのアドバイスと共に研究の整理や論文作成など，熱心なご指導を頂きました 神戸大学大学院 工学研究科 中村 匡秀 准教授に心から深く感謝します.

本研究を進めるに当たり，多くのアドバイスを頂いた 奈良先端科学技術大学院大学 情報科学研究科 ソフトウェア工学講座 大平 雅雄 助教に深く感謝します.

本研究を進めるに当たり，多くの技術的なアドバイスを頂いた 奈良先端科学技術大学院大学 情報科学研究科 ソフトウェア工学講座 森崎 修司 助教に深く感謝します.

本研究を進めるに当たり，多くのアドバイスを頂いた 奈良先端科学技術大学院大学 情報科学研究科 ソフトウェア設計学講座 飯田 元 教授に心から深く感謝します.

本研究を進めるに当たり，多くのアドバイスを頂いた 熊本高等専門学校 情報工学科 神崎 雄一郎 助教に感謝します.

本研究を進めるに当たり，多くの技術的なアドバイスを頂いた 京都産業大学コンピュータ理学部 コンピュータサイエンス学科 玉田 春昭 助教に感謝します.

奈良先端科学技術大学院大学 情報科学研究科 ソフトウェア工学講座 ソフトウェアセキュリティグループのメンバの牛窓 朋義君，岡原 聖君，武田 隆之君の3人には，技術的なご助言やご協力を頂きました．ここに記して謝意します.

本研究を進めるに当たり，発表資料準備などにおいて，ご助言やご協力を頂いた 奈良先端科学技術大学院大学 ソフトウェア工学講座ならびにソフトウェア設

計学講座の皆様に感謝します.

最後に，日頃より私を励まし，支えてくれた家族に心より感謝します.

参考文献

[1] 4C-Entity. *Policy statement on use of content protection for recordable media (CPRM) in certain applications*, 2001. (Available online).

[2] B. Anckaert, M. H. Jakubowski, R. Venkatesan, and K. De Bosschere. Run-Time Randomization to Mitigate Tampering. In *Proc. of Advances in Information and Computer Security*, Lecture Notes in Computer Science Vol. 4752, pp. 153–168, Springer-Verlag, 2007.

[3] 赤井 健一郎, 三澤 学, 松本 勉. ランタイムデータ探索型耐タンパー性評価法. 情報処理学会論文誌, Vol. 43, No. 8, pp. 2447–2457, August, 2002.

[4] E. Barrantes, D. Ackley, S. Forrest, T. Palmer, D. Stefanovic and D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proc. the 10th ACM Conference on Computer and Communications Security (CCS2003)*, pp. 281–289, 2003.

[5] M. Barrett, and C. Thomborson. Frameworks built on the trusted platform module. In *Proc. of 30th International Computer Software and Applications Conference (COMPSAC2006)*, Vol. 2, pp. 59–62, IEEE Computer Society, 2006.

[6] J. C. Benaloh. Secret sharing homomorphisms: keeping shares of a secret secret. In *Proc. of Advanced in Cryptology*, pp. 251–260, 1987.

[7] P. Červeň. *Crackproof your software.* No Starch Press, San Francisco, 2002.

[8] H. Chang, and M. Atallah. Protecting Software Codes by Guards. In *Proc. of Workshop on Security and Privacy in Digital Rights Management 2001*, Lecture Notes in Computer Science Vol. 2320, pp. 160–175, Springer-Verlag, 2001.

[9] B. Chen, R. Morris. Certifying program execution with secure processors. In *Proc. of the 9th conference on Hot Topics in Operating Systems*, pp. 23, May, 2003

[10] S. Chow, H. Johnson, and Y. Gu. Tamper resistant software encoding. United States Patent 6,594,761, Filed 9 June, 1999, Issued 15 July, 2003.

[11] S. Chow, P. Eisen, H. Johnson, and P. van Oorschot. A white-box DES implementation for DRM applications. In *Proc. of the 2nd ACM Workshop on Digital Rights Management (DRM2002)*, Lecture Notes in Computer Science, Vol. 2696, pp. 1–15, Springer-Verlag, 2002.

[12] S. Chow, P. Eisen, H. Johnson and P. van Oorschot. White-box cryptography and an AES implementation. In *Proc. of the 9th International Workshop on Selected Areas in Cryptography (SAC2002)*, Lecture Notes in Computer Science, Vol. 2595, pp. 250–270, Springer-Verlag, 2003.

[13] S. Chow, H. Johnson and Y. Gu. Tamper resistant control-flow encoding. United States Patent 6,779,114, Filed 19 August, 1999, Issued 17 August, 2004.

[14] F. Cohen. Operating system protection through program evolution. Computers & Security, Vol. 12, No. 6, pp. 565–584, 1993.

[15] C. Collberg. SandMark: A Tool for the Study of Software Protection Algorithms. http://sandmark.cs.arizona.edu/

[16] C. Collberg, C. Thomborson and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proc. of ACM Symposium on Principles of Programming Languages (POPL98)*, pp. 184–196, January, 1998.

[17] C. Collberg, and C. Thomborson and D. Low. Breaking abstractions and unstructuring data structures. In *Proc. of IEEE International Conference on Computer Languages (ICCL98)*, pp. 28–38, May 1998.

[18] C. Collberg and C. Thomborson. Watermarking, tamper-proofing, and obfuscation - Tools for software protection. *IEEE Trans. on Software Engineering*, Vol. 28, No. 8, pp. 735–746, 2002.

[19] C. Collberg, C. Thomborson and D. Low. Obfuscation techniques for enhancing software security. United Sates Patent 6,668,325, Filed 9 June 1998, Issued 23 December, 2003.

[20] C. Collberg and J. Nagra. Surreptitious Software. Addison Wesley Professional, 2009.

[21] V. Cortier and G. Steel. Synthesising secure APIs. Institut national de recherche en informatique et automatique (INRIA), inria-00369395, version 1, 2009.

[22] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle and Q. Zhang. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. of the 7th conference on USENIX Security Symposium*, pp. 63–78, 1998.

[23] C. N. Drake. Computer software authentication, protection, and security system. U.S. Patent 6,006,328, Filed July 12, 1996, Issued December, 21, 1999.

[24] H. Etoh and K. Yoda. Protecting from stack-smashing attacks. June, 19, 2000. http://www.trl.ibm.com/projects/security/ssp/main.html

[25] FIPS PUB 140-2. Security Requirements for Cryptographic Modules. National Institute of Standards and Technology, Issued 2001.

[26] C. H. Gebotys. Low energy security optimization in embedded cryptographic systems. In *Proc. of the 2nd IEEE/ACM/IFIP International Conference on Hardware/software codesign and system synthesis*, pp. 224–229, 2004.

[27] C. Del Grossoa, G. Antoniolb, E. Merlob and P. Galinier. Detecting buffer overflow via automatic test input data generation. *Computers & Operations Research*, Vol. 35, Issue 10, pp. 3125–3143, 2008.

[28] D. Grover, The protection of computer software - its tech-nology and applications. The British Computer Society Mono-graphs in Informatics, Cambridge University Press, 1989 (Second Edition 1992).

[29] J. Havrilla. Borland/Inprise Interbase SQL database server contains backdoor superuser account with known pass-word. US-CERT, Vulnerability Note VU#247371, Revision 46, https://www.kb.cert.org/vuls/id/247371, 1 December, 2001.

[30] Hex-Rays. IDA Pro: Disassembler and Debugger, http://www.hex-rays.com /idapro.

[31] W. Hu, J. Hiser, D. Williams, A. Filipi, J. W. Davidson, D. Evans, J. C. Knight, A. Nguyen-Tuong and J. Rowanhill. Secure and practical defense against code-injection attacks using software dynamic translation. In *Proc. of the 2nd Intenational Conference on Virtual execution environments*, pp. 2–12, 2006.

[32] 石間宏之, 亀井光久, 齊藤和雄. ソフトウェアの耐タンパー化技術. 情報処理, Vol. 44, No. 6, 2003.

[33] M. Jacob, D. Boneh and E. Felten. Attacking an obfuscated cipher by injecting faults. In *Proc. of ACM Workshop on Digital Rights Management*, Lecture Notes in Computer Science, Vol. 2696, pp. 16–31, Springer-Verlag, 2003.

[34] Y. Kanzaki, A. Monden, M. Nakamura and K. Matsumoto. Exploiting self-modification mechanism for program protection. In *Proc. of the 27th IEEE Computer Software and Applications Conference*, pp. 170–179, November, 2003.

[35] M. Kuhn. Cipher Instruction Search Attack on the Bus-Encryption Security Microcontroller DS5002FP. *IEEE Trans. on Computers*, Vol. 47, No. 10, pp. 1153–1157, 1998.

[36] Kracker's & BEAMZ, クラッカー・プログラム大全 ～禁断のシリアルナンバー解析テクニック～. データハウス, 2003.

[37] Bil Lewis. Omniscient Debugger. http://www.lambdacs.com/debugger/.

[38] M. LaDue. The Maginot license: Failed approaches to licensing Java software over the Internet, 1997. http://www.geocities.com/securejavaapplets/maginot.html.

[39] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell and M. Horowitz. Architectural support for copy and tamper resistant software. In *Proc. of the 9th International Conference Architectural Support for Programming Languages and Operating Systems*, pp. 168–177, November, 2000.

[40] H. E. Link and W. D. Neumann. Clarifying obfuscation: improving the security of white-box encoding. Cryptology ePrint Archive, Report 2004/025, International Association for Cryptologic Research, 2004.

[41] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proc. of ACM Conference on Computer and Communications Security*, pp. 290–299, 2003.

[42] S. Loureiro and R. Molva. Function hiding based on error correcting codes. In *Proc. of International Workshop on Cryptographic Techniques and Electronic Commerce (CRYPTEC99)*, pp. 92–98, July, 1999.

[43] A. Main and P. C. van Oorschot. Software protection and application security: understanding the battleground. In *Proc. of International Course on State of the Art and Evolution of Computer Security and Industrial Cryptography*, June, 2003.

[44] M. Mambo, T. Murayama and E. Okamoto. A tentative approach to constructing tamper-resistant software. In *Proc. of 1997 New Security Paradigm Workshop*, pp. 23–33, September, 1997.

[45] J. Meyer and T. Downing. Java Virtual Machine. O'Reilly & Associates, Inc., 1997.

[46] J. Meyer. D-Java. http://mrl.nyu.edu/ meyer/jvm/djava/.

[47] J. Meyer. Jasmin Home Page. http://jasmin.sourceforge.net/.

[48] 門田 暁人, Clark Thomborson. ソフトウェアプロテクションの技術動向 (前編) - ソフトウェア単体での耐タンパー化技術. 情報処理, Vol. 46, No. 4, pp. 431–437, April, 2005.

[49] A. Monden, A. Monsifrot, and C. Thomborson. Tamper-resistant software system based on a finite state machine. *IEICE Trans. on Fundamentals*, Vol. E88-A, No. 1, pp. 112–122, January, 2005.

[50] 門田 暁人, 高田 義広, 鳥居 宏次. ループを含むプログラムを難読化する方法の提案. 電子情報通信学会論文誌, Vol. J80-D-I, No. 7, pp. 644–652, July, 1997.

[51] G. Naumovich and N. Memon. Preventing piracy, reverse engineering, and tampering. Computer, Vol. 36, No. 7, pp. 64–71, July, 2003.

[52] T. Ogiso, Y. Sakabe, M. Soshi and A. Miyaji, Software obfuscation on a theoretical basis and its implementation. *IEICE Trans. on Fundamentals*, Vol. E86-A, No. 1, pp. 176–186, 2003.

[53] P. C. van Oorschot. Revisiting Software Protection. In *Proc. of the 6th International Information Security Conference (ISC 2003)*, Lecture Notes in Computer Science, Vol. 2851, pp. 1–13, Springer-Verlag, 2003.

[54] A. Patrizio. Why the DVD hack was a cinch. Wired News, November, 1999. http://www.wired.com/science/discoveries/news/1999/11/32263.

[55] T. Park and K. G. Shin. Soft tamper-proofing via program integrity verification in wireless sensor networks. *IEEE Trans. on Mobile Computing*, Vol. 4, Issue 3, pp. 297–309, 2005.

[56] J. Patarin and L. Goubin. Secret key cryptographic process for protecting a computer system against attacks by physical analysis. United States Patent 6,658,569, Filed 17 June, 1999, Issued 2 December, 2003.

[57] T. Sander and C. Tschudin. Protecting mobile agents from malicious hosts. *Proc. of Mobile Agents and Security*, Lecture Notes in Computer Science, Vol. 1419, pp. 44–60, Springer-Verlag, 1998.

[58] Soot: A Java optimization framework. http://www.sable.mcgill.ca/soot/.

[59] F.-X. Standaert, T. G. Malkin and M. Yung. Unified framework for the analysis of side-channel key recovery attacks. In *Proc. of Advances in Cryptology - EUROCRYPT 2009*, Lecture Notes in Computer Science Vol. 5479, pp. 443–461, Springer-Verlag, 2009.

[60] G. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. AEGIS: Architecture for tamper-evident and tamper-resistant processing. In *Proc. of the 17th Intenational Conference on Supercomputing (ICS2003)*, pp. 160–171, 2003.

[61] P. Syverson. A taxonomy of replay attacks. In *Proc. of the 7th IEEE Computer Security Foundations Workshop*, IEEE Computer Society, pp. 131–136, 1994.

[62] 玉田 春昭, 中村 匡秀, 門田 暁人, 松本 健一. API ライブラリ名隠ぺいのための動的名前解決を用いた名前難読化. 電子情報通信学会論文誌 D, Vol. J90-D, No. 10, pp. 2723–2735, October, 2007.

[63] 玉田 春昭, 中村 匡秀, 門田 暁人, 松本 健一. C 言語におけるライブラリ呼び出し隠蔽のための名前難読化手法. 暗号と情報セキュリティシンポジウム (SCIS2007) 予稿集, January, 2007.

[64] H. Tamada. AddTracer, Injecting tracers into Java class files for dynamic analysis, http://se.naist.jp/addtracer/.

[65] The decompilation Wiki of Program-Transformation.Org, http://www.program-transformation.org/Transform/DeCompilation.

[66] Trusted Computing Group. `http://www.trustedcomputinggroup.org/`.

[67] P. M. Tyma. Method for renaming identifiers of a computer program. United States Patent 6,102,966, August, 2000.

[68] W. E. Vesely, F. F. Goldberg, N. H. Roberts and D. F. Haasl. Fault Tree Handbook. Tech. Rep. NUREG-0492, U.S. Nuclear Regulatory Commission, 1981.

[69] VUPEN Security. Sony PSP Photo Viewer TIFF Image Handling Client-Side Code Execution Vulnerability. http://www.vupen.com/english/advisories/2006/3419.

[70] VUPEN Security. Apple iPhone and iPod touch TIFF Image Remote Code Execution Issues. http://www.vupen.com/english/advisories/2007/3485.

[71] C. Wang, J. Hill, J. Knight and J. Davidson. Protection of software-based survivability mechanisms. In *Proc. International Conference of Dependable Systems and Networks*, pp. 193–202, July, 2001.

[72] やねう解析チーム. 解析魔法少女美咲ちゃん マジカルオープン！. 秀和システム, August. 2004.

[73] 山内 寛己. プログラムの実行系列差分による耐タンパー性評価手法の提案. Master's Thesis, Graduate School of Information Science, Nara Institute of Science and Technology, NAIST-IS-MT0351135, February, 2005.

[74] H. Yamauchi, Y. Kanzaki, A. Monden, M. Nakamura and K. Matsumoto. Software Obfuscation from Crackers' Viewpoint. In *Proc. the IASTED International Conference on Advances in Computer Science and Technology(IASTED ACST2006)*, pp. 286–291, January, 2006.

[75] H. Yamauchi, A. Monden, M. Nakamura, H. Tamada, Y. Kanzaki and K. Matsumoto. A goal-oriented approach to software obfuscation. *International Journal of Computer Science and Network Security*, Vol. 8, No. 9, pp. 59–71, September, 2008.

付録

A. SandMark に実装されている難読化手法

SandMark に実装されている 39 種類の難読化手法は，以下の通りである．

1. Array Folding
   一次元配列を構造の意味を保ちながら，多次元配列に変換する．

2. Array Splitter
   一次元配列を 2 つの配列に分割する．

3. BLOAT
   BLOAT 用いてバイトコードを最適化する．

4. Block Marker
   プログラム中の全ての基本ブロックに無作為に 0 か 1 を付ける．

5. Bludgeon Signatures
   全てのメソッドを Object 型配列で引数を受け取り，返り値を Object 型に変換する．

6. Boolean Splitter
   boolean 型の変数，または，配列を 2 つの変数，配列に分割する．

7. Branch Inverter
   if-else 句の条件を意味を保ちながら，反転させる．

8. Buggy Code
   オリジナルの基本ブロックに影響しないようにランダムに意味のないコード片を挿入する．

9. Class Encrypter
   対象のクラスファイルをあらかじめ暗号化し，実行時にそのクラスファイルを復号して読み込む．

10. Class Splitter
    継承関係を用いてクラスを 2 つのクラスに分割する.

11. Constant Pool Reorder
    コンスタントプールの順序を入れ替える.

12. Duplicate Registers
    元のローカル変数の値が変更される度に新たな変数を生成する.

13. Dynamic Inliner
    実行時にインスタンスメソッドをインライン展開する.

14. False Refactor
    共通の挙動を持たない複数のクラスにスーパークラスを定義し，偽の関連性を持たせる.

15. Field Assignment
    偽のフィールドを割り当てる.

16. Inliner
    クラスメソッドをインライン展開する.

17. Insert Opaque Predicate
    ブール式すべてに恒偽の Opaque Predicate を挿入する.

18. Integer Array Splitter
    メソッド中のローカル変数の配列を 2 つに分割し，その配列に応じて，配列の操作（初期化，読み込み，書き込み，配列の長さの参照など）も修正する.

19. Interleave Methods
    Java 固有のメソッド (e.g. toString()) または，特別なメソッド (e.g. main()) でないメソッドでかつ，シグニチャが同一のメソッドの組を見つけ，メソッド相互に影響がないようにメソッドを結合する.

20. Irreducibility
    メソッドに Opaque predicate を通じて条件分岐を追加して，制御フローを肥大化させる．

21. Merge Local Integers
    2 つの int 型の変数を 1 つの long 型に結合する．

22. Method Merger
    シグニチャが同一のクラスメソッド全てを 1 つの大きいメソッドにまとめる．

23. Objectify
    プリミティブ型でないフィールドをオブジェクト型に置き換える．

24. Opaque Branch Insertion
    Opaque Predicate を利用して，メソッド中にランダムな分岐を挿入する．

25. Overload Names
    メソッドのオーバーライドを利用し，可能な限り同じ名前に変換する．

26. Parameter Alias
    初期化メソッド，抽象メソッド，ネイティブメソッドに 3 つに該当しないメソッドで，かつ，Object 型をパラメータとして受け取るメソッドのパラメータに Thread Local Storage の値を利用する．

27. Promote Primitive Registers
    全ての int ローカル変数に対して，java.lang.Integer に置き換える．

28. Promote Primitive Types
    メソッドごとの全てのプリミティブ型の変数をラッパークラスのインスタンス変数に変更する．

29. Publicize Fields
    フィールドを全てクラスフィールドに書き換える．

30. Random Dead Code
    元のメソッドとは無関係のコード片をメソッドの終端に追加する.

31. Rename Registers
    ローカル変数名をランダムな識別子に書き換える.

32. Reorder Instructions
    メソッド内の基本ブロック間の命令の順序を入れ替える.

33. Reorderer Parameter
    メソッド全ての引数の順序を入れ替える.

34. Simple Opaque Predicates
    恒真の Opaque Predicate を挿入する.

35. Split Classes
    クラスファイルを 2 つに分割する.

36. Static Method Bodies
    非クラスメソッドをヘルパークラスメソッドとスタブに分割する.

37. String Encoder
    プログラム中の文字リテラルを暗号化し, 参照時に復号するように変換する.

38. Transparent Branch Insertion
    手当たり次第メソッドに分岐を挿入する.

39. Variable Reassigner
    メソッド中のローカル変数を再割り当てる.