Return-Oriented Programmingを用いる自己破壊的耐タンパーソフトウェアの検討

大石 和臣[1,a)]

概要：自己破壊的耐タンパーソフトウェアは改変が行われると本来とは異なる動作をすることによって解析・攻撃からプログラムを保護する耐タンパー化技術である．今までに提案されている自己書換えコードに基づく自己破壊的耐タンパーソフトウェアは自己書換えコードが動作しない CPU 上では利用できない．この課題を解決する方法としてデータメモリを利用する構成方法が提案されており，Return-Oriented Programming (ROP) を用いる．本稿では，ROP を用いる自己破壊的耐タンパーソフトウェアについて検討し，その特徴と課題を考察する．

キーワード：耐タンパーソフトウェア, 自己破壊, 自己書換えコード，Return-Oriented Programming (ROP), データメモリ, ソフトウェア保護

A Study on Tamper Resistant Software utilizing Return-Oriented Programming

Kazuomi Oishi[1,a)]

Abstract: Self-destructive tamper resistant software based on self-modifying code was proposed. However, it cannot be used on a CPU which cannot run self-modifying code. As a solution to this shortcoming, another construction method of self-destructive tamper resistant software that utilizes data memory was proposed. The method utilizes Return-Oriented Programming (ROP). In this manuscript, we study the self-destructive tamper resistant software utilizing ROP more in detail and discuss its features and issues.

Keywords: tamper resistant software, self-destruction, self-modifying code, return-oriented programming (ROP), data memory, software protection

1. はじめに

パーソナルコンピュータ（PC），スマートフォン，IoT（Internet of Things）デバイス，インターネットに代表される情報通信処理装置およびネットワークが社会に広く普及している．現在は，ユーザのクレジットカード番号，暗号資産，購買履歴，SNS の利用履歴等の個人に関する情報や，学校・病院・企業・政府・工場・発電所・政府・軍事組織等が扱う社会的に大きな影響を与え得る重要情報等の様々な価値ある情報が PC 等の装置によって処理されている．その結果，それらの情報処理に関するセキュリティが重要性を増しており，暗号を中核とする情報セキュリティ技術が活発に研究開発され現実に多くの装置において利用されている．PC，スマートフォン，IoT デバイス等に暗号等の情報セキュリティ技術がソフトウェア実装されることは一般的である．その場合，攻撃者がそれらの装置を入手して直接解析・攻撃しても，実装されたデータやアルゴリズムを読むことが困難である性質（秘密情報守秘性）や改変することが困難である性質（機能改変困難性）を保つことが求められる．そのような性質を持つソフトウェアは耐タンパーソフトウェア [1] と呼ばれ，今までに多くの研究

[1] 静岡理工科大学 情報学部 コンピュータシステム学科
Shizuoka Institute of Science and Technology, Department of Computer Science
[a)] oishi.kazuomi@sist.ac.jp

開発が行われてきた．

その一つとして，アルゴリズム公開型の自己破壊的耐タンパーソフトウェア [9], [10] が提案されている．これはアンチデバッギング技術と，動的自己書換えによる自己破壊的タンパー応答と，相互依存型自己インテグリティ検証 [3] とを組み合わせた構成を持ち，1 ビットでも命令を改変されると自己破壊的なタンパー応答を連鎖的に発生するため，プログラム非実行型解析（静的解析）とプログラム実行型解析（動的解析）を妨げて改変を防ぐことが可能である．仕様と実装を公開することが可能であり，パラメータを変えることにより解析・攻撃に対する耐性を強めることができる．この自己破壊的耐タンパーソフトウェアは自己書換えコードを用いており，メモリアーキテクチャがフォンノイマン型のコンピュータ，例えば，PC の CPU として普及している Intel 社の x86 上で動作する．フォンノイマン型のメモリアーキテクチャは，コード（命令）を格納する命令メモリとデータを格納するデータメモリが物理的に同じであり，CPU とメモリを接続するバスが一つである．これに対して，命令メモリとデータメモリが物理的に異なり，それぞれに接続する独立なバスを持つメモリアーキテクチャはハーバード型と呼ばれ，フォンノイマン型より命令実行効率が良く，PC より計算資源が少ない組込みシステム向けマイコンに採用されている例がある．ARM は組込みシステム向けマイコンとして多くのスマートフォンや IoT デバイスが採用している．ARM マイコンの一つである Cortex-M3 プロセッサは命令メモリとデータメモリを一部分離している modified Harvard Architecture を採用しており，x86 のように自己書換えコードを利用することは必ずしもできない．この問題を解決するために，自己書換えコードを利用しない自己破壊的耐タンパーソフトウェアとして，データメモリを利用する構成方法 [8] が提案された．本稿は [8] の構成方法を対象とし，その特徴について考察する．

以下，2 で既存の関連研究について述べ，3 でデータメモリを利用する自己破壊的耐タンパーソフトウェアの構成方法を説明する．4 で構成方法の特徴を考察し，最後に 5 でまとめる．

2. データメモリを利用する耐タンパーソフトウェア

今までに提案されている自己破壊的耐タンパーソフトウェアの構成方法について，それらの概要と違いを述べる．

文献 [9], [10] で提案された構成方法（OM 法）で作成される自己破壊的耐タンパーソフトウェアは，実行時に命令メモリの特定の領域を対象とするハッシュ値を計算し，そのハッシュ値に基づき自己書換えを行う．この自己インテグリティ検証の対象は命令メモリであり，プログラムの実行に伴い変化する命令メモリの内容が本来の実行時状態から 1 ビットでも改変されているとハッシュ値が変わるため仕様とは別の自己書換えを実行し，仕様と異なる動作を連鎖的に発生する．つまり，改変に対して自己破壊的タンパー応答を発生し解析・攻撃を防ぐ．

OM 法の提案以降，OM 法を基礎とする，自己書換えコードを用いない自己破壊的耐タンパーソフトウェアの構成方法 [8], [12], [13] が提案された．文献 [12] の方法はデータメモリ上の中間コードと独自のインタプリタを用いるが，その設計・実装と解析の困難さの評価が難しい．文献 [13] の方法は動的呼び出しを用い，スタックの状態をインテグリティ検証の対象に含める特徴を持つが，自己インテグリティ検証に対する汎用的な攻撃であるページ複製攻撃 [14][*1] を防ぐことができないという課題がある．

文献 [8] の方法は以下のアイデアを特徴とし，自己書換えコードを利用せずに自己破壊的耐タンパーソフトウェアを作成できる．

アイデア 1　耐タンパー化対象のホストプログラムに含まれる分岐，つまり jump を indirect jump に変換し，その分岐先アドレスをインテグリティ検証で求めた値に基づいて計算する．

アイデア 2　データメモリの内容をインテグリティ検証の対象とし，プログラム（インテグリティ検証を含む）の実行に伴ってデータメモリの内容を更新し，データメモリの既定とは異なる書換えが行われた場合は正常な処理が行われない．

アイデア 3　プログラムの中で用いられる定数を別の値に変換（偽装）してコードに格納しておき，インテグリティ検証で求めた値に基づいて本来の値に戻して計算に使用する．

アイデア 4　命令メモリのインテグリティ検証とデータメモリのインテグリティ検証を相互に関連付ける．

つまり，ホストプログラムの命令メモリだけでなくデータメモリも保護対象とし，実行に伴い変化するデータメモリの内容も対象とするハッシュ値を計算する．そのハッシュ値に基づき indirect jump の実行あるいは偽装された定数の復元を行うため，改変されると本来とは異なる jump の実行あるいは本来とは異なる定数に基づく計算が行われ自己破壊的タンパー応答を発生する．また，命令メモリのインテグリティ検証とデータメモリのインテグリティ検証を相互に関連づけることで自己インテグリティ検証の相互依存性を従来方法より高めるとともに，ページ複製攻撃を防ぐこともできる．

[*1] ページ複製攻撃は改変した攻撃対象プログラムと未改変の攻撃対象プログラムをメモリに配置し，メモリへのアクセス機能を改造した OS 上で対象プログラムを実行する．OS はメモリアクセスを制御して，命令実行時は改変したプログラムから命令をフェッチし，コード領域のインテグリティ検証時は未改変のプログラムからデータをフェッチする．従って，自己インテグリティ検証は成功しつつ改変したプログラムが実行される．自己書換えコードを用いることでページ複製攻撃に対抗できる [6].

3. Return-Oriented Programming を用いる自己破壊的耐タンパーソフトウェアの構成方法

文献 [8] はアイデア 2 のデータメモリのインテグリティ検証について複数の方法が利用できることを示し，その一つとして ROP を利用する方法 [2] (Parallax) の手法と定数の偽装とを組み合わせた以下の方法を提案している．

3.1 用語の説明

耐タンパー化したいソースプログラム（ホストプログラム）を P，P のアセンブリプログラムを P_{asm}，P が耐タンパー化されたマシン語プログラムを Q とする．

P_{asm} のいくつかの jump は indirect jump に変換されており，Q の実行中に分岐先アドレスが定まる．分岐命令を JI と呼ぶ．

indirect jump の分岐先アドレスを $to\_address$, $to\_address$ を間接的に求めるために用いる（ラベルの）アドレスを $base$，インテグリティ検証の対象領域のハッシュ値を hv とし，それらを関連付ける変数としてマスク $mask$ を導入し，以下のように定義する．

$$to\_address - base = h(hv) + mask \tag{1}$$

ここで，関数 h は，ハッシュ関数の出力をアドレスのビット長以下のビット長のデータに変換する関数である．以降では，説明を簡単にするため，hv のビット長はアドレスのビット長以下，$h(x) = x$ とする．

マシン語プログラム内の命令列（コード）を含む領域をコード領域と呼ぶ．コード領域の n_{jump} 個の分岐命令をアドレスの下位から上位の順に分岐命令 1（JI-1），分岐命令 2（JI-2），...，分岐命令 JI-n$_{jump}$ とし，分岐命令 i（JI-i）の分岐先アドレスを計算するインテグリティ検証ルーチンをインテグリティ検証ルーチン i（IVR-i）とする（$1 \leq i \leq n_{jump}$）．

3.2 作成手順

(1) P あるいは P_{asm} にアンチデバッギングルーチンとハッシュ関数とディジタル署名検証ルーチンを追加する．ディジタル署名の対象領域の先頭アドレスと長さおよびデータ領域内の署名データには仮値を入れておく．

(2) P_{asm} の中から分岐命令 JI を n_{jump} 個選ぶ．

(3) 以下の耐タンパー化アルゴリズムを実行する．

耐タンパー化アルゴリズム

入力：P_{asm}，JI-1，JI-2，...，JI-n_{jump}．

出力：耐タンパー化されたマシン語プログラム Q

(step-1) P_{asm} の分岐命令 JI-i を indirect jump IJI-i に変換する（$1 \leq i \leq n_{jump}$）．なお，IJI-i は 1 個の命令とは限らず，命令列の場合があることに注意する．ROP ガジェットのアドレス（定数）に対して動的に値が求められるように後述のコードを挿入する．

(step-2) IJI-i に対するインテグリティ検証ルーチン IVR-i を以下のように生成する（$1 \leq i \leq n_{jump}$）．

(step-2.1) コード領域とデータ領域から保護対象とする領域を定める．データ領域のうち，非確定的にデータの更新が行われる領域は保護対象に含めない．保護領域のハッシュ値を求めるコードを生成する．保護領域の先頭アドレスと長さおよびマスクには仮値を入れておく．

(step-2.2) $to\_address = base + hv + mask$ を求めるコードを生成する．

(step-3) インテグリティ検証ルーチン IVR-i と分岐先アドレス $to\_address$ を求めるコードを P_{asm} に挿入する（$1 \leq i \leq n_{jump}$）．

(step-4) Parallax の ROP チェイン作成方法に従って，検証関数（verification function）を定め，ホストプログラムに含まれる既存の ROP ガジェットを探索し，それらを制御するように検証関数を ROP チェインに変換する．スタックに置かれる ROP ガジェットのアドレスを動的に求めるための値（後述のマスク）には仮値を入れておく

(step-5) 保護領域を分割（各保護領域の先頭アドレスと長さを決定）し，循環する相互依存の順序を定める．

(step-5.1) インテグリティ検証ルーチンを任意に 1 個選ぶ．アドレス下位から j 番目（$1 < j < n_{jump}$）のインテグリティ検証ルーチン j（IVR-j）とする．

(step-5.2) 保護領域を n_{area} 個の領域に分割する[*2]．

(step-5.3) 循環する相互依存の順序を定める[*3]．

(step-5.4) P_{asm} をアセンブルしたマシン語プログラム Q から各領域の先頭アドレスと長さを求め，Q の仮値と入れ替える．スタックに置かれる ROP ガジェットのアドレスを動的に求めるための値（後述のマスク）を仮値と入れ替える．

(step-6) マスク値を求めて設定する．最初に，j 番目の領域に対して，それを保護領域とするインテグリティ検証ルーチンに関してハッシュ値とマスク値を求め，それらを仮値と置換する．次に，仮値が置換されたインテグリティ検証ルーチンを含む領域に対して，それを保護領域とするインテグリティ検証ルーチンに関し

[*2] 分割に関する詳細な条件は [9] を参照するものとして本稿では省略するが，コード領域内の j 番目の領域はマスクを含まないように分割される．

[*3] 相互依存の順序を定めるための詳細な条件は [9] を参照するものとして本稿では省略するが，j 番目の領域を除き，ある領域に含まれるインテグリティ検証ルーチンはそれが含まれない領域 1 個を対象としてそのハッシュ値を求めるため，n_{jump} 個の領域が数珠つなぎ状の循環を形成する．

てハッシュ値とマスク値を求める処理を同様に行う．以降，全ての領域に対して同様の処理を繰り返し，全ての仮値をマスク値に置換する．

(step-7) ディジタル署名の対象領域に対するディジタル署名を署名生成鍵を用いて計算し，その値をデータ領域の仮値と入れ替える．全ての置換処理を完了した Q を出力する．(end)

4. 考察

文献 [8] では前述の 4 個のアイデアの実現可能性と実装可能性について考察された．本章では，それらの検討をさらに進め，Parallax の手法とその活用方法に焦点を当てて考察する．また実装とセキュリティについても考察する．

4.1 Reterun-Oriented Programming

ROP は，攻撃対象コンピュータの命令メモリにロードされている命令を組み合わせて攻撃プログラムを構成し，それを実行する攻撃手法である [4], [5], [11]．メモリにロードされているアプリケーションや共有ライブラリに含まれるリターン命令 `ret` で終わる短い命令列を ROP ガジェットと呼ぶ．ROP ガジェットを適切に組み合わせることで攻撃対象コンピュータ上で任意の (Turing-complete) 処理を実行できる．ROP ガジェットを適切な順序で実行させるために，ROP ガジェットのアドレスを適切な順序でスタック上に配置する．このスタックを ROP チェインと呼ぶ．ROP チェインに CPU の実行制御を移すことで攻撃処理を起動できる．文献 [5] の ROP ガジェットと ROP チェインの例を図 1 に示す．メインメモリにロードされている `ret` で終わる 2 行の命令列 3 個が ROP ガジェットであり，それらの ROP ガジェットのアドレス 3 個と任意の 2 個の値がスタックに積まれている．

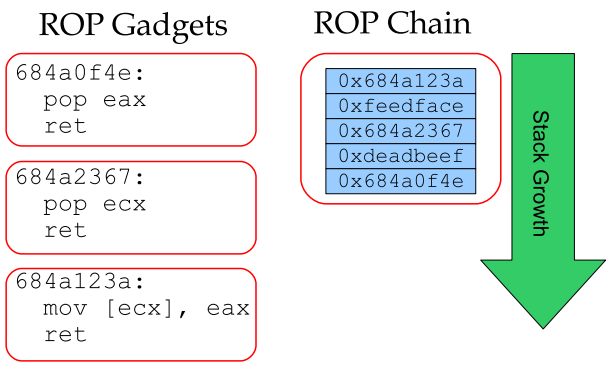

図 1 Return-Oriented Write4 Gadget

この例では，最初にスタックのトップにあるアドレス `0x684a0f4e` に実行制御が移り，そのアドレスの命令 `pop eax` が実行され，スタック・トップから `0xdeadbeef` が pop されてレジスタ `eax` に格納される．次に，命令 `ret` が実行され，スタック・トップのアドレス `0x684a2367` に実行制御が移り，そのアドレスの命令 `pop ecx` が実行され，スタック・トップから `0xfeedface` が pop されてレジスタ `ecx` に格納される．次に，命令 `ret` が実行され，スタック・トップのアドレス `0x684a123a` に実行制御が移り，そのアドレスの命令 `mov [ecx], eax` が実行され，レジスタ `eax` に格納した値 (`0xdeadbeef`) をレジスタ `ecx` にセットしたアドレス (`0xfeedface`) に書き込む．つまり，任意のアドレスに任意の値を書き込めるので任意の処理を原理的に実行できる．

4.2 Parallax

Parallax [2] は ROP を利用して命令メモリのインテグリティ検証を行う．ホストプログラムに ROP を適用し，ホストプログラム本来の命令と ROP に基づくインテグリティ検証の命令とがオーバーラップするように構成する．Parallax の概要を図 2 を参照して説明する．長方形はホストプログラムが格納されているメモリを表し，(1) から (4) の順に処理が施される．

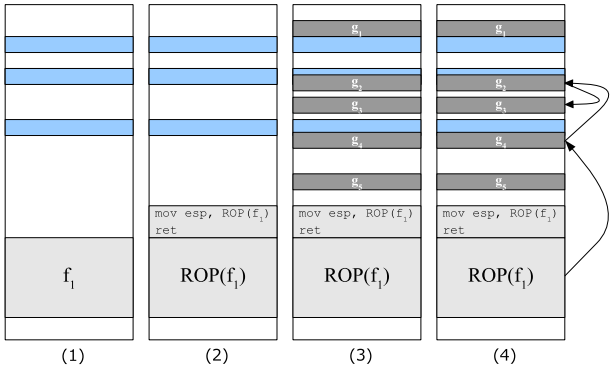

図 2 A high-level overview of Parallax

(1) ホストプログラムから検証関数 f_1 および保護したいコード領域（水色の長方形）を選ぶ．

(2) 検証関数を ROP チェインに変換する．この時点ではガジェットのアドレスは不明のため仮値が指定されている．

(3) ホストプログラムに含まれる ROP ガジェット候補を全て見つけ出す．次に，保護したいコード領域に対してオーバラップする新しいガジェットを付け加えることができるか否かを確認する．付け加えることができるガジェットがあれば付け加える．g_1 から g_5 はそのようにして設けられたガジェットである．

(4) 実際のガジェットを使うために ROP チェインを再コンパイルし，前述の仮値を実際のアドレスの値に更新する．保護したい領域にオーバラップしないガジェットよりオーバラップするガジェットを優先して使用する．

保護したいコード領域の命令に対して改変が行われるとROP に基づく検証命令が影響を受けて正常に実行されず，その結果プログラムは正常に動作しなくなる．つまり，機能改変困難性を持つ自己破壊的耐タンパー技術である．

ROP チェインはスタック，つまりデータメモリに置かれるためページ複製攻撃に対して耐性を持つが，ROP チェインの改変を防ぐ方法が別途必要である．相互依存型自己インテグリティ検証と組み合わせる案が [2] で言及されているがその実装は今後の研究とされていた．

4.3 Parallax の活用方法

文献 [8] では，Parallax を適用することでアイデア 2 と 3 と 4 を実現できると考察されている．それらについてさらに検討する．

前述の作成手順では ROP ガジェットのアドレス値を偽装してデータメモリに格納しておき，命令メモリあるいはデータメモリを対象とするインテグリティ検証で求めた値に基づいて偽装値を本来の値に戻して検証関数の実行に使用する．従って，アイデア 2 と 3 が同時に実現される．この場合，インテグリティ検証対象に改変が行われると本来とは異なる ROP チェイン（内の ROP ガジェットのアドレス値）の命令が実行され自己破壊的タンパー応答を発生する．ROP チェインをインテグリティ検証の対象とすることもアイデア 2 の実現法の一つである．

Parallax によって保護される領域が改変されるとき検証関数は正常に実行されず，データメモリ内の保護対象領域が改変されるときも検証関数は正常に実行されない．つまり，検証関数は命令メモリのインテグリティ検証とデータメモリのインテグリティ検証とを相互に関連付けておりアイデア 4 を実現する．さらに，インテグリティ検証に使うハッシュ関数を Parallax の検証関数とすることも可能であり，これもアイデア 4 の実現法の一つである．

4.4 実装のパラメータ

前述の作成手順は，OM 法と同様にパラメータを変えることにより多様な実装を生み出す．パラメータの例は，

(1) インテグリティ検証 1 回当りの indirect jump 数，
(2) 定数偽装の方法（制御フローに応じた被偽装定数の選択法と再偽装方法），
(3) 検証関数の選択法（検証関数の個数，呼び出し回数等）
(4) 相互依存型自己インテグリティ検証の相互依存の構成方法（複数のインテグリティ検証ルーチンが保護領域を共有するか否か，保護領域が重複を持つか否か），

である．

4.5 実装における課題

OM 法の相互依存型自己インテグリティ検証は命令メモリとデータメモリの全ての領域を保護できる．しかし，データメモリはプログラムの実行に伴い書換えられるため，データメモリのどの領域をいつ検証するかを正確に決めないとプログラムが正しく動作しない．これは実装における課題である．

課題を解決するために OM 法の動的自己書換えを参考にする．動的自己書換えは命令のカムフラージュ [7] に基づき，あらかじめ偽装していたおいた命令を実行時のある期間だけ本来の命令に自己書換えして実行する仕組みである．従って，自己書換えされる命令が実行される実行パスの前後に自己インテグリティ検証結果に基づいて偽装命令を本来の命令に戻す復帰ルーチンと本来の命令を再び偽装命令に戻す隠蔽ルーチンを挿入した．

偽装される定数については，命令のカムフラージュと同様の考え方でインテグリティ検証の挿入位置を定めることができる．一方，データメモリ内のデータについては，その用途によって書換えられるタイミングは異なる．ROP チェイン（内の ROP ガジェットのアドレス値）は検証関数が実行されるときに参照されるため，検証関数の実行時に ROP チェインを対象とするインテグリティ検証を実行することができる．

4.6 セキュリティ

ROP を用いる自己破壊的耐タンパーソフトウェアの機能改変困難性については，Parallax に準ずるインテグリティ検証が付加されているため，動的自己書換えによる自己破壊的耐タンパーソフトウェアと同等以上の強さを持つと考えられる．秘密情報守秘性については，命令のカムフラージュの代わりに定数のカムフラージュを採用しているため，同等程度の強さだと思われる．

5. まとめ

ROP を用いる自己破壊的耐タンパーソフトウェアについて検討し特徴を考察した．ROP を利用するインテグリティ検証方法（Parallax）を適用し，自己書換えコードを用いず indirect jump と ROP に基づいて自己破壊的タンパー応答を実現する．Parallax の適用により命令メモリとデータメモリのインテグリティ検証を相互に関連付ける．実装の課題として，データメモリを対象としてインテグリティ検証を行うがデータメモリのどの領域をいつ検証するかを正確に決める必要がある．

謝辞 本研究は，JSPS 科研費 20K11821 の助成を受けている．

参考文献

[1] D. Aucsmith, “Tamper resistant software: an implementation,” Proceedings of the First International Workshop on Information Hiding, Springer-Verlag, LNCS vol.1174, pp.317–333, 1996.
[2] D. Andriesse, H. Bos, and A. Slowinska, “Parallax: im-

plicit code integrity verification using return-oriented programming," 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2015), pp.125–135, Jun. 2015.

[3] H. Chang, M. J. Atallah, "Protecting software codes by guards," Revised Papers from the ACM CCS-8 Workshop on Security and Privacy in Digital Rights Management, Springer-Verlag, LNCS vol.2320, pp.160–175, 2001.

[4] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-Oriented Programming without Returns," In Proceedings of CCS 2010, pp.559-72. ACM Press, Oct. 2010.

[5] D. A. Dai Zovi, "Practical return-oriented programming," `http://trailofbits.files.wordpress.com/2010/04/practical-rop.pdf`, accessd Aug. 23. 2021.

[6] J. T. Giffin, M. Christodorescu, and L. Kruger, "Strengthening software self-checksumming via self-modifying code," 21st Annual Computer Security Applications Conference (ACSAC), Dec. 2005.

[7] 神崎雄一郎, 門田暁人, 中村匡秀, 松本健一, "命令のカムフラージュによるソフトウェア保護方法, " 信学論（A), vol.J87-A, no.6, pp.755–767, June 2004.

[8] 大石和臣, "データメモリを利用する耐タンパーソフトウェア," 信学技報, ISEC2016-45, Vol. 116, No. 207, pp. 43-48, 2016 年.

[9] 大石和臣, 松本勉, "自己破壊的タンパー応答を発生する耐タンパーソフトウェア," 電子情報通信学会論文誌. A, J94-A(3), pp.192-205, 2011.

[10] K. Oishi and T. Matsumoto, "Self destructive tamper response for software protection," ASIACCS'11, pp.490-496, 2011.

[11] H. Shacham, "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)," CCS '07 Proceedings of the 14th ACM conference on Computer and communications security, pp.552-561 2007.

[12] 吉田直樹, 吉岡克成, 松本勉, "可変な中間コードとして振る舞うデータ部とそれを実行するインタプリタ部からなる 2 部構成の耐タンパーソフトウェア作成法," 情報処理学会論文誌 Vol.55, No.2, pp.1100-1109, 2014.

[13] 渡邉直紀, 吉田直樹, 松本勉, "動的呼び出しを用いてソフトウェアの耐タンパー性を高める方法," 2016 年 暗号と情報セキュリティシンポジウム（SCIS2016), 4C2-3, 2016.

[14] P. C. van Oorschot, A. Somayaji, and G. Wurster, "Hardware-assisted circumvention of self-hashing software tamper resistance," IEEE Trans. on Dependable and Secure Computing, vol.2, no.2, pp.82–92, 2005.