メモリ効率の良い証明可能安全なウィルス検出方式

元橋　颯充[1,a)]　米山　一樹[1]

概要：マルウェアのメモリへの injection からコンピュータを守る手法として、Lipton らは証明可能安全なウィルス検出方式を提案した。しかし、彼らの方式では、元のプログラムを拡張する必要があるため、メモリを通常のプログラム実行よりも多く必要とするという問題点がある。本稿では、Lipton らの方式を基に、よりメモリ効率の良い方式を提案し、既存方式と同様の安全性を満たしていることを示す。

Memory Efficient and Provably Secure Virus Detection

Hayami Motohashi[1,a)]　Kazuki Yoneyama[1]

Abstract: Lipton et al. introduced a provably secure virus detection scheme. However, their scheme needs larger memory than the original program because the program is extended to detect virus injections. In this paper, we introduce a new virus detection scheme which needs smaller memory than Lipton et al.'s scheme. We also prove that our scheme is secure in the same security model of their scheme.

1. はじめに

インターネット環境の普及に伴い、コンピュータは常にネットと繋がり、またネットワークを介したプログラムを実行する機会が増大している。これはすなわちネットワーク越しにマルウェアに感染し、攻撃を受ける機会も増大していることを意味している。よって、コンピュータウィルスやマルウェアの脅威に対して、コンピュータを守ることが重要である。こうした攻撃からコンピュータを守る方法はいくつも考案されてきた。例えば、ソフトウェアを読み込むときに、プログラムのデータや命令セットをランダムな場所に移動するようなコンパイルをしてからプログラムを実行する方法が知られている [4]。これは、攻撃者に間違った path でプログラムを実行させることで、攻撃を防ぐといった考えである。他には、プログラムの命令セットを鍵を使用して暗号化することでランダム化し、それが実行されるときに復号することで正しいプログラムを実行するという防御法がある [5]。この様に、ソフトウェアをマルウェアから守る方法は多く提案されている。しかし、既存方式の多くではマルウェアに対する安全性がフォーマルに証明されていない。Lipton ら [1][2][3] は、暗号理論を応用して、ソフトウェアを実行する際にウィルスの injection の有無を検証する証明可能安全なウィルス検出方式を提案した。彼らはウィルスの injection のパターンに応じたいくつかのレベルの安全性を定式化し、それぞれのレベルの安全性を満たす複数の方式を提案している。しかし、彼らの方式では、injection を検出するためにソフトウェアの実行の際のメモリ使用量が元のプログラムより増えてしまうという問題があった。レジスターサイズを 64 ビット、セキュリティパラメータを 128 とした場合は、元のプログラムのメモリ使用量の 14 倍を必要とする。

本稿では、Lipton らの提案したウィルス検出方式より、メモリ効率の良いウィルス検出方式を提案する。提案方式では、メモリの使用量は Lipton らの方式の約半分になる。具体的には、同条件のパラメータで、元のプログラムのメモリ使用量の 8 倍で済む。Lipton らの方式では、検出するための仕組みを埋め込むために使用するメモリ範囲をあらかじめ確保していたが、その中に実際には安全性上必要ない部分が含まれている。我々は、必要なメモリ量だけを確保することで、メモリ使用量を削減する。また提案方式が、Lipton らの最も強い安全性モデルで安全であることを示す。

2. 準備

2.1 秘密分散共有法

完全な m-out-of-m 秘密共有分散法では、ディーラがある秘密 s を m 個のシェアと呼ばれる $s_1, ., ..s_m$ に符号化し、s_i を参加者に与える。全てのシェアを持つ参加者のみが、

[1] 茨城大学院理工学研究科情報工学専攻 Ibaraki University
[a)] 19NM731X@vc.ibaraki.ac.jp

元の秘密 s を復元することができ、$m-1$ 個のシェアを持つだけでは、秘密 s に関する情報は一切漏れない。

本稿では、秘密情報とシェアが $s, s_i \in S = \{0,1\}^n$、となるような秘密分散共有方式を使用する。秘密の再構成は全てのシェア s_i を用いて $s = \oplus_{i-n}^{m} s_i$ とする。このような再構成法のことを XOR-Sharing と呼ぶ。

2.2 MAC (Message Authentication Code)

MAC は、二つのアルゴリズム $(\mathsf{Mac}, \mathsf{Ver})$ からなる。Mac: $\mathcal{M} \times \mathcal{K} \to \mathcal{T}$ は認証タグ生成アルゴリズムであり、メッセージ $m \in \mathcal{M}$ と鍵 $vk \in \mathcal{K}$ から $\mathsf{Mac}(m, vk) \to tag \in \mathcal{T}$ のように認証タグ tag を生成する。Ver: $\mathcal{M} \times \mathcal{K} \times \mathcal{T} \to \{0,1\}$ は検証アルゴリズムであり、$\mathsf{Ver}(m, tag, vk) = 1$ のとき、メッセージは改ざんされていないとし、受理する。

本稿では、セキュリティパラメータ k と $vk = vk_i || vk_2 \in \{0,1\}^{2k}$ に対して、$\mathsf{Mac}(m, vk) = vk_1 \cdot m + vk_2$ のような方式を考える。ここで、$i \in \{1,2\}$ に対して、$vk_i \in \{0,1\}^k$, $tag \in \{0,1\}^k$ であり、また、もし $|m| < k$ であれば、m の長さが k になるように足りない分を 0 で埋める。

3. RAM とウィルス injection のモデル

本節では、コンピュータのモデルとウィルス injection のモデルの概要を説明する。コンピュータのモデルは一般的な Random Access Machine (RAM) に従うが、プログラムとデータが両方ともに RAM の random access memory 上に書き込まれるような、Random Access Stored Program machine (RASP) を考える。

3.1 Random Access Machine

RAM $\mathcal{R}$ は Random Access Memory (RMEM) と Central Processing Unit (CPU) の 2 つの構成要素からなる。RMEM は $m = poly(k)$ 個のレジスターの配列で作られていて、それぞれのレジスターは L-bit の文字列 (word) を記憶する。CPU は、より小さなレジスターセットを持っており、CPU 全体の保存量はセキュリティパラメータ k に対して線形である。また命令セット $\mathcal{I}$ は、どの命令がレジスター上で行われるか、どんなデータが CPU にロードされ、CPU から出力されるかを定義している。CPU レジスターは読み出し専用入力レジスターと出力レジスターを含んでおり、これがユーザ環境とのインターフェースとなる。プログラムカウンター pc には、次の CPU サイクルで読み込まれるメモリの位置を記憶し、また、それぞれのレジスターには、一意のアドレスが割り当てられ、L-bit 長の word を記憶することができる。

RMEM と CPU はフェッチと実行のサイクルで通信を行っているため、CPU サイクルを RAM 実行のラウンドとしてみなすことができる。各ラウンドでは、CPU は RMEM にアクセスし、RMEM のレジスターの中の word を読み込み、$\mathcal{I}$ からの基本命令を実行する。現代の CPU は複数の命令を一度のラウンドで行うことができるが、ここでは、簡単のため 1 回のラウンドでは 1 つの命令だけを実行できるものとして考える。

次に 2 つの構成要素 RMEM、CPU の詳細について説明する。RMEM は $m = |\mathbf{MEM}|$ 個の words の配列 $\mathbf{MEM}$ であり、$i \in \{0,1,...,m\}$ について $\mathbf{MEM}[i]$ は i 番目の word、すなわち i 番目のレジスターの中身である。同様に、CPU レジスターは word の配列 $\mathbf{REG}$ とする。$\mathbf{REG}[i]$ は i 番目のレジスターの中身である。CPU はベクトル $\mathbf{REG}$ と命令セット $\mathcal{I}$ により、$\mathcal{C}$ =($\mathbf{REG}$,$\mathcal{I}$) として表せる。簡単のため、RAM は CPU $\mathcal{C}$ と RMEM $\mathbf{MEM}$ を用いて、$\mathcal{R}$=($\mathcal{C}, \mathbf{MEM}$) と表す。プロトコル実行時の $\mathcal{R}$ の状態は、CPU と RMEM レジスターの全ての内容を含んだ ($\mathbf{REG}, \mathbf{MEM}$) のベクトルである。安全性の定式化をするために、word サイズ、CPU サイズ、メモリサイズはセキュリティパラメータに依存するものとし、RAM family を $\{R_k\}_{k\in\mathbb{N}} = (C_k = (\mathbf{REG_k}, \mathcal{I}_\mathbf{k}), (\mathbf{MEM_k}))$ とする。

3.2 ソフトウェア実行

RAM $\mathcal{R}$=($\mathcal{C}, \mathbf{MEM}$) 上で実行されるプログラムを、ベクトル $\boldsymbol{W} = (w_0, ..., w_{n-1}) \in (\{0,1\}^L)^n$ とする。それぞれの word は、命令、アドレス、プログラムデータで構成されている。レジスターサイズと $\boldsymbol{W}$ の word サイズは同じであり、プログラム $\boldsymbol{W}$ は各 word に従って実行される。

プログラムの実行は以下のように行なわれる。プログラム $\boldsymbol{W}$ はメモリ $\mathbf{MEM}$ より先頭から順に連続して読み出されて、位置 $j (j \geq |W|)$ には (no-op) が読み出され、メモリは埋められる。ユーザが入力 (s) を与えると、その入力はプログラム $\boldsymbol{W}$ が入っているメモリ部分の次の部分に書き込まれる。入力が与えられると、RAM は実行をプログラムカウンター pc が指し示す MEM の word を CPU にフェッチすることで実行する。pc は実行当初は 0 となっているが、実行を行なうごとにインクリメントされていき、メモリに入っている word を順番に実行する。

プログラム $\boldsymbol{W}$ の性質を以下のように定義する。

定義 1. $\mathcal{R} = (\mathcal{C}, \mathbf{MEM})$ が *RAM* であり、プログラム $\boldsymbol{W} = (w_1, ..., w_n)$ が $\mathbf{MEM}[i_1], ..., \mathbf{MEM}[i_n]$ の位置に書き込まれているとき、いかなる *input* に対しても $\mathcal{R}$ の実行は、$j \neq \{i_1, ..., i_n\}$ の位置に対して読み出しと書き込みを行うことはないならば、このようなプログラム $\boldsymbol{W}$ は $\mathcal{R}$ において、*self-restricted* であるという。

定義 2. *RAM family* $R = \{R_k\}_{k\in\mathbb{N}} = (\{C, \mathbf{MEM_k}\}_{\mathbf{k}\in\mathbb{N}})$ にたいして、プログラム $\boldsymbol{W}$ が *RAM family* $\mathcal{R}$ において、*self-restricted* なプログラムであるならば、$k = poly(|\boldsymbol{W}|)$ の任意の $k \in \mathbb{N}$ にたいしてプログラム $\boldsymbol{W}$ は $\mathcal{R}_k$ において *self-restricted* であるという。

定義 3. *RAM* $\mathcal{R}$=($\mathcal{C}, \mathbf{MEM}$) 上で実行されるプログラムが、実行途中に自己のプログラムを書き換えるような命令がないとき、そのプログラムは、*non-self-modifying structured* なプログラムであるという。

3.3 ウィルスの Injection

ウィルスの RAM に対する攻撃は、ウィルスコード

を挿入する位置を RMEM 上から選択することで行われる。ℓ -word のウィルスは $(\overrightarrow{\alpha}, \boldsymbol{W}) = ((\alpha_0, ..., \alpha_{l-1}), (w_0, ..., w_{l-1}))$ と表現され、$\alpha_i \in \overrightarrow{\alpha}$、$w_i \in \boldsymbol{W}$ はメモリの位置、word をそれぞれ表している。ウィルスの $\mathcal{R}$=$(\mathcal{C}, \mathbf{MEM})$ に対する injection はそれぞれの $\alpha_i \in \overrightarrow{\alpha}$ が指す $\mathbf{MEM}[\alpha_\mathbf{i}]$ に w_i を上書きすることで行われ、ウィルス v は次を満たす。(1) すべての $\alpha_i, \alpha_j \in \overrightarrow{\alpha}$ について $\alpha_i \neq \alpha_j$ である、(2) すべての $\alpha_i \in \overrightarrow{\alpha}$ において $\alpha_i \in \{0, ..., |\mathbf{MEM}| - \mathbf{1}\}$、かつ v は $|\overrightarrow{\alpha}| > 0$ であるなら、ウィルス v は空でない。

CPU のすべてのコントロールをのっとり、すべてのレジスターに読み書きができるような強力なウィルスでも CPU レジスターにウィルスそのものを injection することはできないとする。

4. ウィルス検出方式の安全性モデル

本節では、Lipton らのウィルス検出方式 (VDS) のモデルについて説明をする。VDS では、与えられたプログラム、データに対し、ウィルスが injection されたことを検出できるようにプログラムをコンパイルする。ウィルスの検出はチャレンジ&レスポンスの仕組みを用いて行なわれる。

4.1 VDS モデル

VDS は 5 つのアルゴリズム $\mathcal{V}$ = (Generation, Compile, Challenge, Response, Verify) で構成される。

それぞれは以下のように定義される。またここで、$\overset{\$}{\leftarrow}$ は確率的アルゴリズムの実行を表す。

Generation

Generation は確率的アルゴリズムであり、コンパイル用の鍵 K_c と検証用の鍵 K_v の組を出力する。

$(K_c, K_v) \overset{\$}{\leftarrow}$ Generation

Compile

Compile は確率的アルゴリズムであり RAM $\mathcal{R}$=$(\mathcal{C}, \mathbf{MEM})$、$\mathcal{R}$ のためのプログラム $\boldsymbol{W}$、鍵 K_c を入力とし、同じ CPU $\mathcal{C}$ とメモリサイズ $\tilde{\mathbf{MEM}} = poly(\max\{k, |MEM|\})$ を持った $\tilde{R} = (C, |\tilde{MEM}|)$ に対して、安全なプログラム $\tilde{\boldsymbol{W}}$ を出力する。

$\tilde{\boldsymbol{W}} \overset{\$}{\leftarrow}$ Compile$(R, \boldsymbol{W}, K_c)$

Challenge

Challenge は確率的アルゴリズムであり、鍵 K_v、文字列 $z \in Inp_{\text{chal}} \subseteq \{0,1\}^{poly(k)}$ を入力とし、チャレンジ用の文字列 $c \in Out_{\text{chal}} \subseteq \{0,1\}^{poly(k)}$ を出力する。ここで Inp_{chal},Out_{chal} は Challenge の入出力として取り得る値の集合を表す。

$c \overset{\$}{\leftarrow}$ Challenge(z, K_v)

Response

Response は確率的アルゴリズムであり、文字列 $c \in Out_{\text{chal}}$ と $|\tilde{\mathbf{MEM}}|$ 長の word ベクトル $\tilde{\boldsymbol{W}}$ を入力とし、レスポンス $y \in Out_{\text{Resp}} \subseteq \{0,1\}^{poly(k)}$ を出力する。ここで、Out_{Resp} は Response の出力として取り得る値の集合を表す。

$y \overset{\$}{\leftarrow}$ Response$(c, \tilde{\boldsymbol{W}})$

Verify

Verify は、決定的アルゴリズムであり、鍵 K_v とメッセージ $z \in \text{Inp}_{\text{chal}}$、チャレンジ $c \in \text{Out}_{\text{chal}}$、レスポンス $y \in \text{Out}_{\text{Resp}}$ を入力とし、1bit の $b \in \{0,1\}$ を出力する。

$b \overset{\$}{\leftarrow}$ Verify(K_v, z, c, y)

$b = 1$ であれば、検証を受理する。

4.2 VDS の安全性概念

VDS が満たさなくてはならない安全性概念について説明する。1 つ目の性質は verification correctness であり、RAM が攻撃を受けていないときにチャレンジに対するレスポンスが圧倒的確率で受理されることを表す。

定義 4 (Verification Correctness)**.** $\mathcal{R}$ のための任意のプログラム $\boldsymbol{W}$ において、以下が成り立つならば *VDS* $\mathcal{V}$=*(*Generation, Compile, Challenge, Response, Verify*)* は *verification correct* であるという。

$$Pr\Bigg[(K_c, K_v) \overset{\$}{\leftarrow} \text{Generation} \wedge \tilde{W} \overset{\$}{\leftarrow} \text{Compile}(R, \boldsymbol{W}, K_c) \wedge z \in \text{Inp}_{\text{chal}} \wedge \text{c} \overset{\$}{\leftarrow} \text{Challenge}(x, K_v) \wedge y \overset{\$}{\leftarrow} \text{Response}(c, \tilde{\boldsymbol{W}}) \wedge \text{Verify}(K_v, z, c, y) \neq 1\Bigg] \leq \mu(k)$$

ただし μ は無視できる関数。

2 つ目の性質は Compailation Correctness であり、コンパイルされたプログラムが元のプログラムと同じ計算をすることを表す。つまり 2 つのプログラムがユーザから同じ input(s) を受け取ったときには、次の 2 つの性質を満たさなくてはならない。(1) $\mathcal{R}$ が $\tilde{\boldsymbol{W}}$ を実行したときに $\boldsymbol{W}$ を実行したときと同じ出力をする。(2) $\tilde{\boldsymbol{W}}$ を持つ $\mathcal{R}$ の実行と、元のプログラム $\boldsymbol{W}$ を持つ $\mathcal{R}$ の実行との間に写像が存在する。つまり、$\tilde{\boldsymbol{W}}$ を持つ $\mathcal{R}$ の実行時のある瞬間の $\mathbf{MEM}$ の内容に対して、$\boldsymbol{W}$ を持つ $\mathcal{R}$ の実行時における同じ $\mathbf{MEM}$ の内容からの写像が存在する。

$\tilde{\boldsymbol{W}}$ の実行は、$\boldsymbol{W}$ の実行が終了する CPU サイクルの回数とは異なる回数を行った後に終了する。これはコンパイラが $\boldsymbol{W}$ の命令の実行をいくつかの CPU サイクルで行うように拡張するからである。この追加された round の回数は、$\boldsymbol{W}$ が実行を終了するのに必要な round 回数について多項式である。また RAM $\mathcal{R}$=$((\mathbf{REG}, \mathcal{L}), \mathbf{MEM})$, $p \in \mathbb{N}$ において、$RMEM(\boldsymbol{W}, p, \overrightarrow{x} = (x_1, ..., x_i))$ は入力 $x_1, ..., x_i$ でプログラム $\boldsymbol{W}$ を実行したとき p 番目の CPU サイクルの終了時の $\mathbf{MEM}$ の状態を表している。また、起こりえる $\mathcal{R}$ のメモリ状態の集合を Σ_{RMEM} によって表す。加えて、$\mathcal{R}_{\text{out}}(\boldsymbol{W}, p, \overrightarrow{x} = (x_1, ..., x_i))$ で、p 回目の round が終わったときの最初に出てくる出力を示す。

定義 5 (Q-bounded emulation)**.** 写像 $Q : \mathbb{N} \to \mathbb{N}$ を計算可能な単一増加の関数とする。任意の $p \in \mathbb{N}$、入力 $\overrightarrow{x}$ において、計算可能な関数 $\mathcal{J} : \Sigma_{\tilde{R}MEM} \to \Sigma_{RMEM}$ が次

の二つの性質を持つ時、$|\tilde{\mathbf{MEM}}| = \mathrm{poly}(\mathbf{MEM})$ である *RAM* $\mathbf{M\tilde{E}M} = (\mathcal{C}, \tilde{\mathcal{R}})$ におけるプログラム $\tilde{\boldsymbol{W}}$ は $\boldsymbol{W}$ の *Q-bounded emulation* という。

$$\tilde{\mathcal{R}}_{\mathrm{out}}(\tilde{\boldsymbol{W}}, Q(p), \overrightarrow{x}) = \mathcal{R}_{\mathrm{out}}(\boldsymbol{W}, p, \overrightarrow{x})$$
$$\mathcal{J}(\tilde{\mathcal{R}}_{\mathbf{MEM}}(\tilde{\boldsymbol{W}}, Q(p), \overrightarrow{x})) = \mathcal{R}_{\mathbf{MEM}}(\boldsymbol{W}, p, \overrightarrow{x})$$

定義 6 (compilation correctness)**.** コンパイルされたプログラム $\tilde{\boldsymbol{W}}$ が無視できる確率を除いて、$\boldsymbol{W}$ の *Q-bounded emulation* であるような、既知の関数 $Q : \mathbb{N} \to \mathbb{N}$ をもつ *RAM* $\mathcal{R}$ のプログラム $\boldsymbol{W}$ ならば、*VDS* $\mathcal{V}$ は *RAM* $\mathcal{R}$ において *compilation correct* であるという。

VDS の中でユーザはチャレンジを特別な入力として受け取り、それを検証しなくてならない。そのために要求される性質が self-responsiveness である。これはコンパイルされたプログラム $\tilde{\boldsymbol{W}}$ が Response アルゴリズムでレスポンスを計算できるコードを含んでいるということである。これにより、特別な入力 $x' = (check, c)$ を $\tilde{\boldsymbol{W}}$ が受け取ったときの出力はウィルスを injection されていない場合は、圧倒的確率で、入力 c と等しくなくてはならない。

定義 7 (self-responsiveness)**.** 任意のプログラム $\boldsymbol{W}$ と入力 $(check, c)$ にたいして *round* ρ の実行時に入力 $(check, c)$ が入力レジスターに書き込まれた場合に、以下の性質を満たすような単一増加関数 $Q : \mathbb{N} \to \mathbb{N}$ が存在する時、*VDS* は *RAM* $\mathcal{R}$ で *self-responsiveness* であるという。

$$Pr\Bigg[(K_c, K_v) \xleftarrow{\$} \mathsf{Generation} \wedge \tilde{\boldsymbol{W}} \xleftarrow{\$} \mathsf{Compile}(R, \boldsymbol{W}, K_c) \wedge$$
$$z \in Inp_{\mathrm{chal}} \wedge c \xleftarrow{\$} \mathsf{Challenge}(z, K_v) \wedge y \xleftarrow{\$} \mathsf{Response}(c, \tilde{W}) \wedge$$
$$y \neq \tilde{\mathcal{R}}_{\mathrm{out}}(\tilde{\boldsymbol{W}}, Q(p), (check, c))\Bigg] \leq \mu(k)$$

4.3 VDS のセキュリティゲーム

前節では、VDS によって保護されているソフトウェアがウィルスに攻撃されていないときの動作について説明した。本節では,RAM 上にウィルスを injection することを目的とする攻撃者 Adv と、それを検出することを目的とするチャレンジャー Ch の間のセキュリティゲームにより、VDS の安全性の定義を示す。

セキュリティゲームを $\mathcal{G}_{VDS}^{\mathcal{R},\mathcal{V},W}$ で表し、以下のように実行する。

チャレンジャー Ch はアルゴリズム Generation を実行し、鍵のペア (K_c, K_v) を得る。そして、アルゴリズム Compile で RAM $\mathcal{R}$ のプログラム $\boldsymbol{W}$ を新しいプログラム $\tilde{\boldsymbol{W}}$ にコンパイルする。Ch は $\mathcal{R}$ 上でプログラム $\tilde{W}$ の実行を仮想実行、すなわち、CPU サイクルを仮想実行し、与えられた時点での全体の状態を保持する。攻撃者 Adv は攻撃者自身の判断でメモリ **MEM** 上の好きな位置にウィルスを inject することが許されている。続いて、Ch はウィルス検出を実行する。アルゴリズム Challenge でチャレンジ c を計算し、その後入力 $(check, c)$ を仮想実行している RAM に与え、レスポンス y を計算させる。

VDS にとって、最も強い攻撃ケースを考えるため、攻撃者は仮想実行中の好きなタイミングでウィルスを injection することができ、ウィルスが injection した後からチャレンジを受け取るまでに、RAM を何 round 実行するかについても攻撃者が決めることができる。攻撃者は検出プロセスが行われる前までの round 回数 ρ_{pre}、攻撃者が injection を行いたい round ρ_{att} を決定することができる。さらに、攻撃者 Adv は元のプログラム $\boldsymbol{W}$ を知っていて、それに対して、いかなる入力も決め、知ることができるとすることで、攻撃者が元のプログラム $\boldsymbol{W}$、$\mathcal{R}$ への入力に関して、どれくらいの情報を保持しているかが分からない状態にする。

定義 8. *The repeated attack game* $\mathcal{G}_{VDS}^{\mathcal{R},\mathcal{V},W}$

(1) Adv は連続した入力 $\overrightarrow{x} = (x_1, ..., x_i)$ を選択し、ベクトル $\overrightarrow{\rho}_{\mathrm{pre}} = (\rho_{\mathrm{pre}}^{(1)}, ..., \rho_{\mathrm{pre}}^{(\mu)})$ 、また $i \in \{1, ..., \mu\}$、で $\rho_{\mathrm{pre}}^{(i)} = poly(k)$、$\rho_{\mathrm{att}} = poly(k)$ と、加えて、$virus = (\overrightarrow{\alpha} = (\alpha_1, ..., \alpha_v), \tilde{\boldsymbol{W}} = (w'_0, ..., w'_v))$ とともに、$(\overrightarrow{x}, \overrightarrow{\rho}_{\mathrm{pre}}, \rho_{\mathrm{att}}, virus)$ を *Ch* に送る。

(2) Ch は $K \xleftarrow{\$} \mathcal{K}$、$\tilde{\boldsymbol{W}} \xleftarrow{\$} \mathsf{Compile}(R, \boldsymbol{W}, K)$ を実行する。

(3) Ch は以下のステップを $i = 1, ..., \tau$ まで連続して行う。

(a) Ch は続いて、$\rho_{pre}^{(i)}$ *round*、入力 $\overrightarrow{x}$ で $\tilde{\mathcal{R}} = (\mathcal{C}, \mathrm{MEM})$ の仮想実行を行い、カウンター ρ が $\rho = \rho_{\mathrm{att}}$ になった場合は、*Ch* はウィルスの *injection* を仮想実行する。ウィルスが何もしないものであれば、*Ch* は $\mathbf{M\tilde{E}M}$ をそのままにし、そうでなければ、それぞれの $l \in \{0, ..., v\}$ で *Ch* は $\mathbf{M\tilde{E}M}[\alpha_l]$ に $w'_l \in \boldsymbol{W}'$ を書き込む。$\tilde{\mathcal{R}}$ が $output(s)$ を出力した場合は、それを *Adv* に渡す。

(b) Ch は以下の様に検出プロセスを行う。

(i) Ch は $z_i \in \mathrm{Inp}_{\mathrm{chal}}$、$c_i \xleftarrow{\$} \mathsf{Challenge}(z_i, K)$ を行う。

(ii) Ch は $(check, c_i)$ を入力レジスターに書き込み、$Q(\rho_{\mathrm{pre}})$ の追加 *round* 分 $\tilde{\mathcal{R}}$ の実行を行う。もし、*round* カウンター ρ が $\rho = \rho_{\mathrm{att}}$ であるなら、*Ch* はウィルスの *inject* を行い、ウィルスが空でなく、有効である場合にそれぞれの $l \in \{0, ..., v\}$ で *Ch* は $\mathbf{M\tilde{E}M}[\alpha_l]$ に $w'_l \in \boldsymbol{W}'$ を書き込む。

(c) $\tilde{\mathcal{R}}$ が出力レジスターに y を出力したら、*Ch* は $b_i \xleftarrow{\$} \mathsf{Verify}(K, z_i, c_i, y_i)$ を計算する。

VDS の安全性定義は以下の通りである。

定義 9. 次の二つの性質を満たす場合、*RAM family* $\mathcal{R}$ における *VDS* $\mathcal{V}$ = (Generation, Compile, Challenge, Response, Verify) は安全である。

(1) $\mathcal{V}$ は *verification correct,compilation correct,self-responsive* である。

(2) 空でない有効なウィルスを *injection* する、任意の多

項式時間攻撃者 Adv において、$Pr[b_i = 1] \leq \mu(k)$ を満たす。ただし、ステップ *(3)* において、$i = j - 1$ で *injection* が行われたとする。

5. 提案方式

本節では、Lipton らが提案した continuous injection をしないウィルスに対する VDS を基にしてメモリの使用量を減らすような新しい VDS 方式を提案し、安全性証明を与える。ここで、continuous injection とはウィルス $(\vec{\alpha}, \boldsymbol{W}) = ((\alpha_0, ..., \alpha_{l-1}), (w_0, ..., w_{l-1}))$ において、$w_0||...||w_{l-1} \in \{\bot\}^*||\{0,1\}^*||\{\bot\}^*$ で表すことができるウィルスを injection することである。またウィルスのサイズは非 $\bot$ である bit の数であるとする。このような continuous injection を行なわないウィルスはソフトウェアが入ったメモリ上に飛び飛びに injection することが可能になる。Lipton らの方式では、このようなウィルスを検出するために耐漏えい公開鍵暗号 [6]、ハッシュ関数 (ランダムオラクル)、MAC を用いている。彼らの方式では、検出のために元のプログラムを拡張し、公開鍵暗号の秘密鍵や MAC の認証タグなどをメモリ上に埋め込むため、元のプログラムより多くのメモリを必要とする。しかし、彼らの方式におけるプログラム拡張は、安全性を満たすためには必要のないメモリ領域まで含んでいる。具体的には、彼らの方式では MAC の認証タグを 2 つ埋めこめるようにメモリ領域を確保していたが、MAC の認証タグは実際には 1 つのみ埋め込んでいた。そのため、認証タグを格納するために確保していた部分が使用されず、残ってしまっていた。そこで、プログラム拡張時に、安全性証明に必要なメモリ量だけを確保することで、メモリ使用量を削減した方式を提案する。

以下、提案方式を示す。

アルゴリズム Generation

(1) 耐漏えい公開鍵暗号の鍵のペアを 2 つ $(PK_{od}, SK_{od}), (PK_{ev}, SK_{ev})$ 生成する。秘密鍵の鍵長を k ビットとする。

(2) 秘密鍵 Sk_{od}, SK_{ev} をそれぞれ
$SK_{od} = SK_{od,1}, SK_{od,2}, \ldots, SK_{od,\frac{n}{3}}$,
$SK_{ev} = SK_{ev,1}, SK_{ev,2}, \ldots, SK_{ev,\frac{n}{3}}$
のように分割する。ここで $n = \frac{3K}{L}$ であり、また $i \in \{1, \ldots \frac{n}{3}\}$ において $|SK_{od,i}| = |SK_{ev,i}| = L$ である。

(3) コンパイル用の鍵 K_c を次のように構成する。
$K_c = SK_{od,1}||SK_{ev,1}|| \ldots ||SK_{od,\frac{n}{3}}||SK_{ev,\frac{n}{3}}$.

(4) 2 つの秘密鍵で構成されたコンパイル用の鍵 K_c、検証用の鍵 $K_v = (PK_{od}, PK_{ev})$ を出力する。

$\boldsymbol{W}$ への $\boldsymbol{W}_{Resp}$ の追加

プログラム自身が、特定のチャレンジを入力として受け取ったときにレスポンスを返すことができるように、プログラムに、Response プロセスを行うプログラム $\boldsymbol{W}_{resp}$ を元のプログラムの最後に付加する。これら 2 つのプログラムがどちらも self-restiriced であるならば、このプログラムは RMEM を修正することなく、今までに考えてきたモデルでの実行が可能である。また $\boldsymbol{W}_{resp}$ の付加により、プログラム $\boldsymbol{W}$ の実行中に、ある input$(check, c)$ を受け取ると、プログラム $\boldsymbol{W}$ は実行を中断し、$\boldsymbol{W}_{Resp}$ の実行を行なうようになる。$\boldsymbol{W}_{Resp}$ の実行が完了するとすぐに、元の $\boldsymbol{W}$ の実行の続きを始める。この動作について説明する。

プログラムはまず入力レジスターに書き込まれている最後の文字列が $(check, c)$ $(c \in \{0,1\}^k)$ であるかを確かめる。確認の結果、最後の文字列が $(check, c)$ でなかった場合、プログラムは通常実行される。最後の文字列が指定の形をしていた場合は、プログラムは以下のように動作を行う。

(1) CPU 全体の状態を、メモリの使用していない部分の先頭部に保存する。
(2) $\boldsymbol{W}_{Resp}$ が入っているメモリのブロックまで jump し、$\boldsymbol{W}_{Resp}$ の実行、すなわちレスポンスの計算を行う。
(3) $\boldsymbol{W}_{Resp}$ の実行が終われば、ステップ 1 で保存しておいた CPU の状態に戻す。

$\boldsymbol{W}_{Resp}$ を付加することでプログラムは self-responsiveness を満たす。$\boldsymbol{W}_{Resp}$ の付加は、以下に記すプロセス Spread で行なわれる。またプロセス Spread は Compile の中で用いられるサブルーチンである。

Spread プロセス

(1) $\boldsymbol{W}$ の最後に $\boldsymbol{W}_{resp}$ の埋め込みを行なう。
(2) 初期化：$i = 0, ..., (n+2)|\boldsymbol{W}| - 1$、$\tilde{w}_i$ =no_op を入れる。
(3) $i = 0, ..., |\boldsymbol{W}| - 1$ について
- $w_i = (\mathsf{jumpby}, z)$ であれば、$w'_i := (\mathsf{jumpby}, (n+2)(z+1) - 1)$
- $w_i = (\mathsf{jumpby\_if}, P, j, z)$ であれば、$w'_i := (\mathsf{jumpby\_if}, P, j, (n+2)(z+1) - 1)$
- $w_i = (\mathsf{read}, i', j)$ であるとき、$w_{i'} \notin \{(\mathsf{jumpby}, , ,), (\mathsf{jumpby\_if}, , ,)\}$ であれば、$w'_i = (\mathsf{read}, (n+2)i', j)$、そうでなければ $w'_i = (\mathsf{read}, (n+2)i' + 1, j)$
- $w_i = (\mathsf{write}, j, i')$ であれば、$w'_i := (\mathsf{write}, j, (n+2)i')$
- w_i が上記のどれでもない場合は、$w'_i = w_i$

(4) $i = 0, ..., |W| - 1$ について
- $w_i \notin \{(\mathsf{jumpby\_if}, , ,), (\mathsf{jumpby\_if}, , ,)\}$ であれば、$\tilde{w}_{(n+2)i} = w'_i$,$w'_{(n+2)i+1} := (\mathsf{jumpby}, n)$
- そうでなければ、$\tilde{w}_{(n+2)i} = (\mathsf{jumpby}, 0)$,$\tilde{w}_{(n+2)i+1} = w'_i$

(5) $\tilde{\boldsymbol{W}} = (\tilde{w}_0, ..., \tilde{w}_{|\boldsymbol{W}|(n+2)-1})$ を出力する。

アルゴリズム Compile では、秘密鍵のシェアを Spread により拡張されたプログラムの部分に格納するが、ほかにも MAC の計算を行い、生成した tag も格納する必要がある。MAC 認証タグの計算は、$(\mathsf{write\_auth}, j, i)$ プロセスで行う。$(\mathsf{write\_auth}, j, i)$ の詳細は次の通りである。

(write_auth,j,i) プロセス

$n = \frac{3k}{L}$ であることに注意する。

(1) メモリから以下を読み出す

- $X = \mathbf{REG}[j], J = \mathbf{MEM}[i+1]$
- $vk_1 = \mathbf{MEM}[i+2]||...||\mathbf{MEM}[i+\frac{n}{3}+2]$
- $vk_2 = \mathbf{MEM}[i+\frac{n}{3}+3]||...||\mathbf{MEM}[i+\frac{2n}{3}+3]$

(2) ハッシュ関数 H を用いて、$t_i = \mathsf{Mac}(X||J,(H(vk_1),H(vk_2)))$ を計算する。その後、$t_i = t_{i,1}||...||t_{i,\frac{n}{3}}$ と分割をする。$l \in \{1,...,\frac{n}{3}\}$ において、$t_{i,l} \in \{0,1\}^L$ となる。

(3) $\mathbf{MEM}[i]$ に X を書き込む。

(4) $l = 1,...,n$ で $\mathbf{MEM}[i+\frac{2n}{3}+3+l]$ に $t_{i,l}$ を書き込む。

プロセス Spread、(write_auth, j,i) を使い、アルゴリズム Compile を以下のように構成する。

アルゴリズム Compile

コンパイラは入力として、RAM $\mathcal{R}$, 元のプログラム $\boldsymbol{W} = (w_0,...,w_{|\boldsymbol{W}|-1})$ と鍵 K_c を受け取る。

初期化：$n = \frac{3k}{L}$

(1) Spread を呼び出し、プログラムを拡張する。$\tilde{\boldsymbol{W}} = (\tilde{w}_0,...,\tilde{w}_{|\boldsymbol{W}|(n+2)-1}) \leftarrow \mathsf{Spread}(\boldsymbol{W},n)$

(2) 鍵をシェアに分割する。

(a) $(i,j) \in \{0,...,|\boldsymbol{W}|-2\} * \{1,...,\frac{n}{3}\}$ のような i,j でシェア $SK_{od,j}^{i,i+1}, SK_{ev,j}^{i,i+1} \xleftarrow{\$} \{0,1\}^L$、そして、$\tilde{w}_{i(n+2)+2j+1} = H(SK_{od,j}^{i,i+1}), \tilde{w}_{i(n+2)+2j+2} = H(SK_{ev,j}^{i,i+1})$

(b) $j \in \{1,...,\frac{n}{3}\}$ において、

$\tilde{w}_{(|\boldsymbol{W}|-1)(n+2)+2j+1} = H(SK_{od,j} \oplus_{i=0}^{|W|-2} SK_{od,j}^{i,i+1})$

$\tilde{w}_{(|\boldsymbol{W}|-1)(n+2)+2j+2} = H(SK_{ev,j} \oplus_{i=0}^{|W|-2} SK_{ev,j}^{i,i+1})$

(3) (write_auth, j,i) を実行する

(4) $\tilde{\boldsymbol{W}}$ を出力する。

またこのコンパイラは、self-restricted で non-self-modifying-structured なプログラムを同じく self-restricted で non-self-modifying-structured なプログラムに変換するものである。コンパイルにより、元のプログラム $w_0||w_1||\ldots||w_{|\boldsymbol{W}|-1}||$ は、$w_0||(\mathsf{jumpby},n)||K_{od}^{0,1}||K_{ev}^{0,1}||tag_1||w_1||\ldots||$ のようなプログラムとなる。

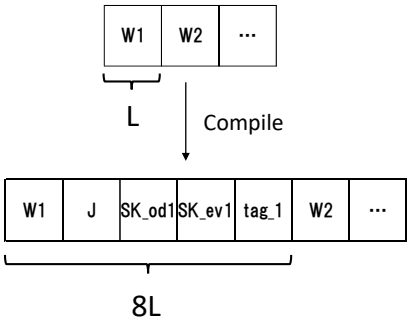

図 1 Compile の例 ($k = 2L$ の場合)

Spread により、$n \in \mathbb{N}$ において、Spread 後のプログラムは n 個の空な (no_op で埋められた) メモリを持つ。Spread によりコンパイルされたプログラム $\tilde{\boldsymbol{W}}$ は実行時に、$\boldsymbol{W}$ と同じ実行手順で動作をするために、word の実行後、次の word を呼び出すために (jumpby, n) を行うことになる。よって元のプログラムが 1 round の実行をするとき、コンパイルされたプログラムは 2 round の実行をすることとなる。

Spread について、以下の補助定理が示されている [1][2][3]。

補助定理 1. プログラム $\boldsymbol{W}$ が、*RAM* $\mathcal{R} = (\mathcal{C}, MEM)$ において、*self-restricted* で *non-self-modifying structured programming* であるならば、任意の $n \in \mathbf{N}$ に対してアルゴリズム $\mathsf{Spread}(\boldsymbol{W},n)$ を行ったときの出力は、*RAM* $\tilde{\mathcal{R}} = (\mathcal{C}, M\tilde{E}M)$ での *self-restricted* で *non-self-modifying structured* なプログラム $\tilde{\boldsymbol{W}} = (\tilde{w}_1,....,\tilde{w}_i)$ である。また $|\mathbf{M\tilde{E}M}| = n|\mathbf{MEM}|$ であり、プログラム $\tilde{\boldsymbol{W}}$ は以下の性質を満たす。

(1) $\tilde{\boldsymbol{W}}$ は $\boldsymbol{W}$ の *Q-bounded emulation* であり、$Q(p) = 2p$ である。

(2) $\tilde{\mathcal{R}}$ 上での $\tilde{\boldsymbol{W}}$ の実行はメモリ位置 $\mathbf{M\tilde{E}M}[i]$ を参照する。ただし、$i \notin (0,1)$ かつ $i \bmod (n+2)$ に対しては、読み出しと書き込みのアクセスを行なわない。

アルゴリズム Challenge

Challenge には、入力として公開鍵 (PK_{od}, PK_{ev}) とランダムな平文 $z \in \{0,1\}^k$ が与えられ、チャレンジ $c = \mathsf{Enc}_{PK_{od}}(\mathsf{Enc}_{PK_{ev}}(z))$ を出力する。

アルゴリズム Response では、チャレンジに対するレスポンスを計算するだけでなく、MAC の検証プロセス (read_auth, i,j) を行う。(read_auth, i,j) は、メモリの i 番目の word とそれに一致する鍵の MAC を検証し、もし検証が成功したならば、word を CPU に残したままにする。検証が失敗した場合は、メモリから少なくとも 1 つ以上の鍵のシェアを消去する。これにより、以降の検証でウィルスが injection されたことが検出される。(read_auth, i,j) の詳細は次の通りである。

(read_auth,i,j) プロセス

$n = \frac{3k}{L}$ であることに注意する。

(1) メモリから以下を読み出す。

- $X = \mathbf{MEM}[i], J = \mathbf{MEM}[i+1]$
- $vk_1 = \mathbf{MEM}[i+2]||...||\mathbf{MEM}[i+\frac{n}{3}+2]$
- $vk_2 = \mathbf{MEM}[i+\frac{n}{3}+3]||...||\mathbf{MEM}[i+\frac{2n}{3}+3]$
- $t_i = \mathbf{MEM}[i+\frac{2n}{3}+4]||...||\mathbf{MEM}[i+n+4]$

(2) $b = \mathsf{Ver}(X||J,t_i,(H(vk_1),H(vk_2)))$

(3) $b = 1$ であるときは、$\mathbf{REG}[J]$ に X を書き込む。その後、命令 I を実行する。それ以外の場合は、$l = 1,\ldots,\frac{2n}{3}+2$ について $\mathbf{MEM}[i+1+l] = 0^L$ とする。出力レジスターに"invalid mac"と出力する。

MAC の検証を含んだアルゴリズム Response は次のように動作する。

アルゴリズム Response

チャレンジ c とコンパイル後のプログラム $\tilde{\boldsymbol{W}}$ を入力とする。

(1) ハッシュ関数 h を用いて $y_h = h(\tilde{\boldsymbol{W}}||c)$ を出力する。その後 CPU から y_h を消去する。

(2) $(\mathsf{read\_auth}, i, j)$ プロセスを実行し、MAC の検証を行う。

(3) 鍵の再構成 SK_{od}

(a) $j \in \{1, ..., \frac{n}{3}\}$ において $SK_{od,j} = \oplus_{i=0}^{|W|-1} \tilde{w}_{i(n+2)+2j+1}$ を計算する。

(b) $SK_{od} = SK_{od,0}||SK_{od,1}||\ldots||SK_{od,\frac{n}{3}}$

(4) SK_{od} を使った復号:

$c' = \mathsf{Dec}_{SK_{od}}(c)$ を計算し、c' をレジスター $\mathbf{REG}[0]$ に保存する。

(5) c' を除くすべての CPU レジスターの中を消去する。

(a) すべての $j \in \{1, \ldots, |\mathbf{REG}|\}$ において $\mathbf{REG}[j] = 0$ とする。

(b) 前のステップで実行された (write, j, i) が入った $\mathbf{MEM}[j]$ を $\mathbf{MEM}[j]$ =no_op とする。

(6) 鍵の再構成 K_{ev}

(a) $j \in \{1, ..., \frac{n}{3}\}$ において $SK_{ev,j} = \oplus_{i=0}^{|W|-1} \tilde{w}_{i(n+2)+2j+2}$ を計算する。

(b) $SK_{ev} = SK_{ev,0}||SK_{ev,1}||\ldots||SK_{ev,\frac{n}{3}}$

(7) SK_{ev} を使った復号:

$y = \mathsf{Dec}_{SK_{ev}}(c')$ を計算し、y を出力する。

(8) $y'_h = h(\tilde{\boldsymbol{W}}'||c)$ を出力する。このとき、$\tilde{\boldsymbol{W}}'$ はその時の RMEM の状態を示している。

アルゴリズム Verify

Verify アルゴリズムは入力として、公開鍵 (PK_{od}, PK_{ev})、平文 z、チャレンジ c、Response の出力 (y, y_h, y'_h) をとる。$(z = y) \wedge (y_h = y'_h)$ であるときに,$b = 1$ を出力し、それ以外の場合では $b = 0$ を出力する。

定理 1. *RAM* $\mathcal{R}$ が $\mathcal{R} = \{\mathcal{R}_k\}_{k\in\mathbb{N}} = \{C, \mathbf{MEM_k}\}_{\mathbf{k}\in\mathbb{N}})$ を満たすような *RAM family* であり、耐漏えい暗号が秘密鍵の $(n-1)/n$ *bit* を知っている攻撃者に対して、*CPA* 安全であるなら、*continuous* でない *injection* を行う攻撃者がウィルスをプログラム $\tilde{\boldsymbol{W}}$ の入ったメモリ $\boldsymbol{MEM}$ の任意の場所に *injection* したとしても、*VDS* はランダムオラクルモデルの下で、安全である。ただし、実行されるプログラムはすべて *non-self-modifying-structured* なプログラムである。

証明 1. *(*定理 *1)* の証明

安全性 *verification correctness*、*compilation correctness*、*self-responsiveness*、ゲーム $\mathcal{G}_{VDS}^{\mathcal{R},\mathcal{V},W}$ での安全性をそれぞれ証明する。

compilation correctness/self-responsiveness

プログラム $\boldsymbol{W}$ は *RAM* $\mathcal{R} = (\mathcal{C}, MEM)$ において、*self-restricted* かつ *non-self-modifying structured* なプログラムであり、任意の $n \in \mathbf{N}$ で $\mathsf{Spread}(\boldsymbol{W}, n)$ を行ったときの出力は、*RAM* $\tilde{\mathcal{R}} = (\mathcal{C}, M\tilde{E}M)$ における *self-restricted* で *non-self-modifying structured* なプログラム $\tilde{\boldsymbol{W}} = (\tilde{w}_1, ..., \tilde{w}_i)$ である。この時、補助定理 *1* よりコンパイルされたプログラム $\tilde{\boldsymbol{W}}$ は、*q-bounded emulation* であり、$Q(p) = 2p$ であることから、*compilation correctness* は満たす。

また、プログラム $\boldsymbol{W}_{resp}$ は $input(check, c)$ が読み込まれたときにアルゴリズム Response を実行するプログラムであるため、*self-responsiveness* を満たす。

verification correctness

補助定理 *1* より、コンパイルされたプログラム $\tilde{\boldsymbol{W}}$ の実行時は鍵のシェアが入っているメモリにアクセスしないことが保証できる。またアルゴリズム Verify の実行時は、鍵のシェアの入っているメモリに対し、書き込みを行わない。よって、*RAM* $\mathcal{R}$ が $(check, c)$ を受け取り、プログラム $\boldsymbol{W}_{Resp}$ がアルゴリズム Response を実行する場合、耐漏えい公開鍵暗号が完全性を満たしているならば、二重暗号化されたランダムな平文 z を正しく復号できる。したがって Verify は *1* を返すため、*verification correctness* を満たす。

ゲーム $\mathcal{G}_{VDS}^{\mathcal{R},\mathcal{V},W}$ での安全性

VDS を破ろうとする攻撃者 *Adv* は *continuous* でない *injection* を行うことができるので、*word*、鍵、*tag* に対して上書きをすることができる。次の *2* つのパターンの *injection* に分けて考える。

Case1: 鍵の入っている部分に injection を行う攻撃者

この時、攻撃者はウィルスを少なくとも一つ以上の鍵のシェア、すなわち、SK_{od},SK_{ev} のシェアの少なくとも *1* ブロックに *injection* を行う。その場合に、チャレンジに対し、正しいレスポンスを返すことができないことを証明する。

VDS の検証を破ることができる攻撃者 *Adv* の存在を利用して、耐漏えい公開鍵暗号の *CPA* 安全性を破る攻撃者 *Adv'* を構成する。

Adv' は *Adv* に $\mathcal{G}_{VDS}^{\mathcal{R},\mathcal{V},W}$ の実行を始めさせる。このゲーム内では、*Adv'* は $\mathcal{G}_{VDS}^{\mathcal{R},\mathcal{V},W}$ 内のチャレンジャー *Ch* の役割を担う。*Ch* は、*Adv* から $virus = (\alpha, (w'_1, ..., w'_i))$ を受け取る。α は *virus* が *injection* される位置を表す。*Adv'* は α の情報をゲーム内で *virus* が上書きする鍵のシェアの特定に使用する。具体的には、$i^* = min_{i\in\{\alpha,...,\alpha|v\}}\{i|i \notin \{0,1\}(\mathrm{mod}\ n)\}$ とし、$i^*_n := i^* \mathrm{mod}\ n$ を定義する。また i^* は奇数であると仮定する。ここで i^* が偶数だった場合は、以下の証明を対照的にすることで同じように考えることができる。

CPA 安全性ゲームにおけるチャレンジャー *Ch'* は秘密鍵と公開鍵のペア (SK_{od}, PK_{od}) を生成する。ここで SK_{od} は $SK_{od} = SK_{od,1}, ..., SK_{od,\frac{n}{3}}$ と分割することができる。*Adv'* は $\tilde{SK}_{od,i^*_n} = 0^L$ とする。*Adv'* は秘密鍵漏えい情報を利用して、*Ch'* から $SK_{od,1}, ..., SK_{od,i*-1}$, $SK_{od,i*+1}, ..., SK_{od,\frac{n}{3}}$ を受け取り、$i \in (1, ..., n) \setminus \{i^*_n\}$ において $\tilde{SK}_{od,i} = SK_{od,i}$ とする。すなわち $\tilde{SK}_{od} = \tilde{SK}_{od,1}, ..., \tilde{SK}_{od,i_n*-1}, \tilde{SK}_{od,i_n*}, \tilde{SK}_{od,i_n*+1}, ..., \tilde{SK}_{od,n}$ となる。

Adv' は、異なる鍵のペア $(\tilde{SK}_{ev}, \tilde{PK}_{ev})$ を生成し、

$\tilde{SK}_{ev} = \tilde{SK}_{ev,1}, ..., \tilde{SK}_{ev,\frac{n}{3}}$ のように分割する。*Adv'* は $\tilde{SK} = \tilde{SK}_{od,1}||\tilde{SK}_{ev,1}||...||\tilde{SK}_{od,\frac{n}{3}}||\tilde{SK}_{ev,\frac{n}{3}}$ とする。

鍵 $\tilde{SK}$ を用いて、*Adv'* は *Adv* から受け取った入力に従い、仮想実行を行う。*round* ρ_{pre} 分、*Adv'* は、プログラムを実行する。実行時には検出プロセスが繰り返し行われるが、各検出プロセスのそれぞれにおいて、*Adv* はランダムな平文 x を選択し、チャレンジ $c = \mathsf{Enc}_{\tilde{PK}_{od}}(\mathsf{Enc}_{\tilde{PK}_{ev}}(x))$ を作成し、検出プロセスで使用する。

round ρ_{att} になったら、*Adv'* は *virus* を *injection* し、*round* ρ_{att} 後で、初めに検出プロセスが行われるまで、仮想実行を続ける。検出プロセスが起こったときに、*Adv'* にランダムに x_0, x_1 を選択し、$m_0 = \mathsf{Enc}_{\tilde{PK}_{ev}}(x_0)$,$m_1 = \mathsf{Enc}_{\tilde{PK}_{ev}}(x_1)$ を計算する。*CPA* ゲームの *Ch'* に渡すことで、*Adv'* は $c_b = \mathsf{Enc}_{\tilde{PK}_{od}}(m_b)$ を受け取る。c_b を検出プロセスにおける Challenge の出力として、*RAM* 実行の仮想実行を続け、もし $y = x_{b'}$ になれば、出力として b' を出力する。それ以外は b' をランダムに選び出力する。

ウィルスの *injection* は、検出プロセスを行う前に行われるか、または検出プロセスを行なっているときに行うかの 2 つのパターンがある。後者のパターンは、*CPU* で SK_{od} が消去される前に SK_{ev} のシェアに *injection* を行うケースと SK_{od} が消去された後に SK_{od} のシェアに *injection* を行うケースにさらに分けられる。SK_{ev} のシェアに *injection* を行う場合では、SK_{ev} による復号を行う前に鍵のシェアを上書きしているので、検出プロセスを突破することができなくなる。SK_{od} のシェアに *injection* を行う場合では、その *round* に行われている検出プロセスを突破することはできるが、次に行われる検出プロセス時には、鍵のシェアを上書きしている状態となっているため、検出プロセスを突破することができなくなる。

次に検出プロセスを行う前に *injection* を行う *Adv* が無視できない確率で攻撃に成功したとき、同じく *CPA* ゲームの攻撃者 *Adv'* も無視できない確率で成功することを示す。ウィルスの *injection* が起こる前までは、鍵のシェアが上書きされることはなく、すべてのチャレンジとレスポンスも検出プロセスを満たす。ウィルスが *injection* されると少なくとも一つの鍵のシェアが書き換わることになるが、SK_{od}, i_n^* が書き換わった鍵のシェアであるとき、ここに *injection* された値が入っているか、すべてが 0 の *word* が入っているかは攻撃者の出力に違いを生むことはない。もし *Adv* がチャレンジに正しく答えることができた場合、*Adv'* もまた正しく *bit b* を推定することができる。したがって、*VDS* が安全でない、すなわち、*Adv* が無視することのできない確率で存在する時、*Adv'* もまた無視できない確率で存在する。したがって、耐漏えい公開鍵暗号の *CPA* 安全を破ることができる攻撃者が存在しないとき、検出プロセスを行う前に *injection* を行い *VDS* を破ることができる攻撃者も存在しない。

Case2: 鍵以外の部分にも injection を行う攻撃者

VDS を破ろうとする攻撃者 *Adv* が鍵以外の部分、すなわち、*word*、jumpby、 *SK* のシェア、タグの一部のビットに *injection* をし、*VDS* の検証を突破しようとする場合を考える。

MAC の認証タグは $t_i = w_i||\mathsf{jumpby}_i \times H(vk_1) + H(vk_2)$ のように生成される。*word*、jumpby、*SK* のシェア、タグの一部分に *injection* をし、*VDS* の検証を突破しようとする場合を考える。*injection* が行われた後の認証タグは $t_i' = w_i'||\mathsf{jumpby}_i' \times H(vk_1') + H(vk_2')$ となる。この時 vk_1', vk_2' はハッシュ関数 H に入力されるので、*Adv* は *injection* により、*SK* のシェアの一部を上書きしても *MAC* 鍵の $(H(vk_1'), H(vk_2'))$ の値をコントロールすることができない。そのため、$t_i' = w_i'||\mathsf{jumpby}_i' \times H(vk_1') + H(vk_2')$ を満たすような組み合わせは 2^k 通り存在し、*MAC* の偽造ができる確率は $\frac{1}{2^k}$ である。

以上より、*word*、jumpby、*SK* のシェア、タグの一部のビットに *injection* をしたとしても検証を突破することができる確率は高々 $\frac{1}{2^k}$ であり、このような *injection* を行い、*VDS* を突破できるような攻撃者は無視できる確率でしか存在しないことが示された。

□

参考文献

[1] Richard J.Lipton, Rafail Ostrovsky, Vassilis Zikas. Provable Virus Detection:Using the Uncertainly Principle to Protect Against Malware. IACR Cryptology ePrint Archive 2015: 728 (2015)

[2] Richard J.Lipton, Rafail Ostrovsky, Vassilis Zikas. Provably secure virus detection. U.S. Patent (pending), Application No. 62/054,160, 2014.

[3] Richard J. Lipton, Rafail Ostrovsky, Vassilis Zikas. Provably Secure Virus Detection: Using The Observer Effect Against Malware. ICALP 2016: 32:1-32:14 (2016)

[4] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, Dan Boneh.On the Effectiveness of Address-Space Randomization. In ACM Computer and Communication Security Symposium, 2004.

[5] Gaurav S. Kc, Angelos D. Keromytis, Vassilis Prevelakis.Countering Code-Injection Attacks With Instruction-Set Randomization. CCS ’ 03: Proceedings of the 10th ACM conference on Computer and communications security, New York, NY, USA, pp.272280, ACM Press, 2003.

[6] Y.Dobis, Y.T.kalai, C.Peikert, V.Vaikuntanathan. Public-key encryption schemes with auxiliary inputs.In TCC 2010, volume 5978 of LNCS, pages 361381. Springer, 2010.