能動的な攻撃者に対するメモリアクセスパターン保護方式の安全性評価

仲野有登[1,2]　披田野清良[1]　清本晋作[1]　櫻井幸一[2]

概要：著作権保護などのためにコンテンツが暗号化されている場合，再生ソフトは復号のための鍵をメモリから入手する必要がある．悪意のあるユーザはメモリアクセスを解析することで鍵がメモリ上のどのアドレスに格納されているかを特定することが可能であり，効率的に鍵を復元することが可能である．このようにアプリケーションがメモリに対して実行するメモリアクセスのパターンを解析することによって鍵などの重要な情報の漏洩につながる場合がある．また，OpenSSL の Heartbleed 脆弱性による情報漏えいのように意図せず情報が流出する可能性がある．これらの脅威への対策として Oblivious RAM(ORAM) が適用可能であり，耐タンパソフトウェアを実現するための要素技術としても注目されている．しかし，ORAM はメモリに格納されているデータを改変することができる能動的な攻撃者は想定しておらず，能動的な攻撃者に対して十分な安全性を確保できない可能性がある．本稿では，メモリ上のデータを改変可能な攻撃者に対して ORAM がどの程度の安全性を有するかを評価し，対策の有効性を検討する．

キーワード：Oblivious RAM，メモリアクセスパターン保護，能動的な攻撃，安全性評価

Security Evaluation of Memory Access Pattern Protection against Active Adversaries

Yuto Nakano[1,2]　Seira Hidano[1]　Shinsaku Kiyomoto[1]　Kouichi Sakurai[2]

Abstract: Video and music files are sometimes distributed to the users with encrypted form, and the secret key for the decryption is embeded into the player application. When the ligitimate users consume the content, the player application read the secret key which is stored on RAM. A molisious user with an ability to observe RAM can extract the secret key by analysing the behaviour of the player. Moreover, a software bug can cause an unintended information leak as shown in Heartbleed. Oblivious RAM (ORAM) is a solution for protecting such information, however, security of ORAM schemes are evaluated against passive adsersaries where they can only have read access to RAM. In this paper, we evaluate the security of ORAM schemes against active adversaries where they alter data on RAM and try to efficiently extract the secret information. We also propose countermeasures against active adversaries.

Keywords: Oblivious RAM, memory access pattern protection, active adversary, security evaluation

1. はじめに

メモリに対するアクセスのパターンから様々な情報が漏洩することが示されており，プログラム保護の観点からパターンを保護することが望まれる．著作権保護などのためにコンテンツが暗号化されている場合，そのコンテンツを視聴するためには再生ソフトによって復号しながら視聴する必要がある．この時，再生ソフトは復号のための鍵をメモリから入手する．悪意のあるユーザがこの再生ソフトのメモリアクセスを解析することで鍵がメモリ上のどのア

[1] KDDI 総合研究所情報セキュリティグループ
Information Security Laboratory, KDDI Research
[2] 九州大学大学院システム情報科学研究院
Faculty of Information Science and Electrical Engineering, Kyushu University

© 2017 Information Processing Society of Japan

ドレスに格納されているかを特定することが可能であり，攻撃者は効率的に鍵を復元することが可能である．メモリアクセスのパターンを保護可能な手法としては *Oblivious RAM (ORAM)* が一般に知られている [4], [5]．プログラムがRAMにアクセスする際に，ORAMを介して行うことで，メモリを監視している攻撃者に対して，プログラムの実行時間以外の情報を秘匿することが可能である．さらに，OpenSSLのHeartbleed脆弱性による情報漏えいなどでメモリ上のデータが流出したとしても，流出したデータから意味のある情報を取り出すことを困難にすることが可能になるほか，メモリ上に展開されているデータを攻撃者から保護できる特徴を利用することで，耐タンパソフトウェアを構成するための要素技術としても利用が考えられる．著作権保護に適用した事例としては文献 [9] などがある．

ORAMの構成には2種類あり，1つは *square root solution* と呼ばれる．*square root solution* では，秘密の規則に従って全データを並び替え，アクセスしたデータは "shelter" と呼ばれる領域に一時的に保管される．もう1つの構成手法は *hierarchical solution* と呼ばれる．*hierarchical solution* では，データが階層的に保存され，上位から下位に向かってアクセスが行われる．square root solution の方が単純であるが，hierarchical solution に比べるとオーバヘッドが大きい傾向にある．どちらの構成もオーバヘッドが大きく，高速化のための手法が多く提案されている [1], [2], [6], [10], [13], [16]．さらに，CPUをクライアント，RAMをサーバとみなし，サーバに対してアクセスのパターンを秘匿することが可能であるため，クラウドへの適用が期待されており，これに関する研究も盛んに行われている [7], [8], [11], [18], [19], [20]．

ORAMの高速化に関する多くの研究によって，クラウド向けには実用的なオーバヘッドに抑えられるようになってきているが，プログラム保護への適用を考えた場合，更に軽量な手法が必要である．そこで，筆者らは新しい手法を提案した [12]．この手法では，ORAMのsquare root solutionと同様にメインのメモリに比べて小さいが安全なメモリ (バッファ) を追加し，ここに直近でアクセスされたデータを一時的に保管しておく．バッファが一杯になった際にメインメモリにデータを退避させるが，この時に退避したデータのアドレスを履歴表に記録していく．この履歴を活用したダミーのアクセスを実施することで，軽量かつ安全なパターン保護を実現している．

ORAMも筆者らの方式も受動的な攻撃者を想定しており，攻撃者はメモリを監視するのみで，メモリ上のデータを変更することは考慮していない．しかし，実際はメモリを監視可能な攻撃者はデータ改変も可能な能動的な攻撃者であることが多く，能動的な攻撃に対する安全性評価が必要である．そこで，本稿では，攻撃者がメモリ上のデータを改変可能だと仮定した場合にメモリアクセスパターン保護が安全であるかどうかの評価を行う．

2. 関連研究

本節では，本稿と関連のある研究としてORAMの一つであるsquare root solutionを紹介した後，ORAMの改良手法であるPath ORAMを紹介する．

2.1 Oblivious RAM(square root solution)

ORAMは攻撃者がメモリにアクセスできる状況においてもソフトウェアを保護可能とするための技術として，Goldreich [4], [5] によって提案された．構成手法として，*hierarchical solution* と *square root solution* の2つが提案されている．ここでは，本稿の内容により近いsquare root solutionについて説明する．square root solutionではもとの n 個のデータに対して，*shelter* と呼ばれる $\sqrt{n}$ 個のデータを保管できる領域を追加する．更に，$\sqrt{n}$ 個のダミーブロックを追加する．その後，$n+\sqrt{n}$ 個のデータに対して，プログラムのみが知る規則 π に応じて並び替えを行う．これによってアドレス i のデータはアドレス $\pi(i)$ に保管されることになるプログラムがアドレス i に保管されているデータに対してアクセスを行う場合，まず，shelter内の全データに対してアクセスを行い，そのデータが含まれるかどうかを判定する．そのデータが含まれていない場合，$\pi(i)$ に対してアクセスを行い，データをshelterにコピーする．そのデータが含まれている場合，$\pi(d+t)$ にアクセスを行い，$(d+t)$ に保管されるデータをshelterにコピーする．ここで，d はダミーのアドレス，t は何回目のアクセスかを表すカウンタである．各アクセスで一つのデータをshelterにコピーするため，$\sqrt{n}$ 回のアクセスでshelterは一杯になる．その場合，新しい規則 π' を選択し，再度データをすべて並び替える．このデータの並び替えは，攻撃者にわからないように行う必要があり，非常に大きなオーバヘッドとなる．

2.2 Path ORAM

ORAM (hierarchical solution, square root solution) はオーバヘッドが大きいという課題があり，多くの改良手法が提案されている [1], [2], [6], [10], [13], [16]．特に軽量な手法としてPath ORAMが提案されている [17]．Path ORAMはメモリ上のデータを2分木として管理し，*bucket* と呼ばれる各ノードが Z 個のデータブロックを保持する．また，セキュアな領域を持っており，そこに *stash* と *position map* を用意する．stashは C ブロックのデータを格納可能であり，Path ORAMがアクセスしたデータを保持する．position mapは各データと格納されている葉を関連付けて保持している．Path ORAMがデータ a にアクセスする場合，position mapから a が葉 l に格納されていることを確認する．葉 l に向かう枝にアクセスし各bucketに保存され

© 2017 Information Processing Society of Japan

ている真のデータを stash に追加する．この操作でデータ b が stash に移動するので b にアクセスを行う．アクセス後，新しい葉 l' を選択し position map を更新する．stash から葉 l' に向かう枝の各 bucket に stash 内のデータを書き出す．bucket に空きがある場合はダミーデータを格納する．

元の $ORAM_0$ が利用する position map のサイズが大きくセキュアな領域に格納しきれない場合は $ORAM_0$ の position map を $ORAM_1$ 内に実装する．さらに $ORAM_1$ の position map を $ORAM_2$ 内に実装するというように再帰的に ORAM を構成することでセキュアな領域に格納が必要な position map のサイズを圧縮する手法を提案している (recursive Path ORAM)．

Path ORAM への完全性確認 Recursive Path ORAM において，メモリ上のデータを改変可能な攻撃者は，高い確率で a) 同じデータに対する繰り返しアクセスと b) 異なるデータへのアクセスを判別することが可能であることが示されている [14], [15]．能動的な攻撃者に対して安全に recursive Path ORAM を構成するため，Ren らは定義 1 のように安全性を定義している．

定義 1. *ORAM* に対して変更を加えることが可能な攻撃者が，検知されることなく *ORAM* の出力を変更することはできない，または，メモリアクセスパターンに関する情報を入手できない場合，安全な完全性確認と呼ぶ

定義 1 を満たす手法として Merkel Tree を利用した完全性確認を提案している．提案手法ではルートハッシュを安全な領域に保持しておき，各 bucket 内の真のデータに対してアクセスを行う前に各データのハッシュ値を計算し，データが改変されていないことを確認する．

3. 準備

3.1 アクセスパターン

文献 [12] と同じく，アクセスパターン保護を次のように定義する．CPU には，小さいが安全なメモリがあり，RAM には安全ではないが，十分なサイズの保存領域があるとする．CPU がメモリ上のデータにアクセスする場合，アドレス i に保管されているデータを読み込む，もしくはアドレス j に値 x を書き込むのいずれかであるとする．この操作をそれぞれ `read`(i)，`write`(j,x) と書く．アクセスパターン保護によって，データの保護だけでなく，アクセスのパターンも保護可能である．データの保護については，安全な probabilistic 暗号を使うことで達成できる．パターンの保護については，一連のアクセス系列がアクセスしたデータの個数以外の情報を攻撃者に与えないようにしたい．これを実現するために，まず，`read` と `wirte` の識別ができないようにする必要がある．これは各アクセスで常に `read` と `wirte` を実行することで実現できる．この場合，アクセスパターン $A(y)$ の識別困難性は，アクセスするデータに対応するインデックス i の識別困難性に置き換える事ができる．自明な手法は，各アクセスですべてのデータにアクセスすることであるが，大きなオーバヘッドが生じる．

3.2 筆者らのアクセスパターン保護方式

筆者らは，RAM に対するアクセスパターンを保護するための実用的な手法を提案し，2 つの実施例を示した [12]．我々の方式の特徴は，アクセスの履歴を保管しておき，履歴を活用したダミーのアクセスを実施することで，オーバヘッドの原因となるデータの並び替えを行わずにアクセスパターンを保護するという点である．図 1 に示すように，安全な領域を追加し，その中にデータを一時的に保管するバッファ $\mathcal{M}$ とアクセスの履歴を保存する領域 $\mathcal{H}$ を設ける．また，データをランダムに並び替えるための規則 π もこの領域に保存する必要がある．

プログラムが起動されると，まず，規則 π にしたがってデータの並び替えを行う．ORAM ではこの並び替えを定期的に行なう必要があるが，この方式では起動時に一度並び替えを行なうだけであるため，オーバヘッドを小さくすることができる．並び替えが完了した後，プログラムはデータへのアクセスを開始する．各アクセスごとに RAM とバッファ間でデータの交換を行い，アクセスが行われる．バッファには M ブロックのデータと対応するアドレスを保管可能でる．バッファのサイズは限られているため，一定アクセス後にいっぱいになる．そうするとランダムに選択されたデータが RAM の元あったアドレスに戻される．この時，どのアドレスにあったデータが RAM に戻されたかを履歴として保存する．履歴表には H アドレスを保管可能であり，いっぱいになるとランダムに選択したアドレスが新しいアドレスで上書きされる．

プログラムがアドレス i_a に保管されているデータ a にアクセスする場合を例に取り，方式の説明を行う．アクセスの度に 2 つのブロックをバッファに移動し，代わりに 2 つのブロックを RAM に移動する．どのブロックをバッファに移動するかは，次のように場合分けされる．

(1) '$a' \in \mathcal{M}$ であれば，(r,p) を移動
(2) '$a' \notin \mathcal{M}$，かつ，'$i'_a \in \mathcal{H}$ であれば，(r,a) を移動
(3) '$a' \notin \mathcal{M}$，かつ，'$i'_a \notin \mathcal{H}$ であれば，(a,p) を移動

ここで，r はランダムに選択したデータ，p は履歴表に含まれるデータを表す．このように選択することで，攻撃者からは，片方はランダムに選択されたためにバッファに移動されているかのように見え，もう片方は履歴表に登録されているためバッファに移動されているかのように見える．条件 2 の場合，ランダムに選択されたブロック r と必要なブロック a がバッファに移動する．ブロック r は (実際そうであるが) ランダムに選択されたダミーブロックのように見える．また，ブロック a はアドレスが履歴に登録されているから (実際にはそうでないが) ランダムに選択された

© 2017 Information Processing Society of Japan

Algorithm 1 メモリアクセスパターン保護

```
1: M をスキャンし 'a' が含まれるかどうかを確認
2: if 'a' ∈ M then
3:    M 内のブロックをランダムに 2 つ選択し，これらを H と L からそれぞれランダムに選択したブロックと交換
4: else
5:    H をスキャンし 'i_a' が含まれるかどうかを確認
6:    if 'i_a' ∈ H then
7:       M 内のブロックをランダムに 2 つ選択し，これらを L から選択したランダムなブロックと 'a' と交換
8:    else
9:       M 内のブロックをランダムに 2 つ選択し，これらを 'a' と H からランダムに選択したブロックと交換
10:   end if
11: end if
12: ℓ_h + 2 個から ℓ_h 個を選択し，H を更新
13: 'a' に対してアクセス
```

かのように見える．疑似コードを Algorithm 1 に示す．

安全性評価プログラムが時刻 t にデータ 'a' に対して行うアクセスを $X_t = a$ と記載するとする．この時，m と ℓ_h をそれぞれバッファ $\mathcal{M}$ と履歴テーブル $\mathcal{H}$ のサイズとすれば，δ ステップ後において $p_M = \Pr[a \in \mathcal{M}] \geq \left(\frac{m-2}{m}\right)^\delta$ と $p_H = \Pr[a \in \mathcal{H}] \geq \left(\frac{\ell_h}{\ell_h+2}\right)^\delta$ が成り立つ．

メモリに対するアクセス $X_i = a_i$ $(i = 1, \ldots, N)$ を監視可能な攻撃者を $\mathcal{A}$ とすれば，保護方式の安全性は $X_t = X_{t+\delta} = a$ となる確率を導出することで可能であるから，

$$
\begin{aligned}
\Pr[X_t = a, X_{t+\delta} = a] &= \Pr[X_t = a] \cdot \Pr[X_{t+\delta} = a] \\
&\leq \frac{1}{m-2}(\Pr[X_{t+\delta} = a \mid \text{case 1}] \cdot \Pr[\text{case 1}] \\
&\quad + \Pr[X_{t+\delta} = a \mid \text{case 2}] \cdot \Pr[\text{case 2}] \\
&\quad + \Pr[X_{t+\delta} = a \mid \text{case 3}] \cdot \Pr[\text{case 3}]) \\
&\leq \frac{p_M}{(m-2)^2} + \frac{(1-p_M)p_H}{(m-2)^2} + \frac{(1-p_M)(1-p_H)}{2(m-2)}.
\end{aligned}
$$

であり，

$$
\epsilon \leq \frac{p_M}{(m-2)^2} + \frac{(1-p_M)p_H}{(m-2)^2} + \frac{(1-p_M)(1-p_H)}{2(m-2)}.
$$

を得る．

4. メモリの内容を変更可能な攻撃者に対する耐性評価

ORAM も筆者らの方式も攻撃者はメモリを監視するのみで，メモリ上のデータを変更することは考慮していない．しかし，実際はメモリを監視可能な攻撃者はデータも改変可能であることが多く，能動的な攻撃者に対する安全性を考慮することは重要である．

ORAM も筆者らの方式もプログラムが行うメモリアクセスのパターンを攻撃者から秘匿するためにダミーが重要な役割を果たしている．ダミーはプログラムの動作に影響を与えないため，メモリ上のデータを自由に変更可能な攻撃者はデータを改変して動作に影響が出るかどうかでダミーを特定可能である．本節では能動的な攻撃者に対するメモリアクセスパターン保護方式の安全性を評価する．表 1 に評価結果をまとめる．

表 1　評価結果

手法	攻撃者がダミーを特定可能な確率
square root	$\frac{\sqrt{n}p_S}{n+\sqrt{n}}$
筆者らの方式	$\frac{1+p_M}{2} \cdot \frac{d}{n+d}$

4.1 攻撃手法

攻撃者として，攻撃対象のアプリケーションが利用するメモリ領域へのアクセスが可能な攻撃者を想定する．また，攻撃者はアクセス可能な領域に格納されているデータを自由に変更可能であるとする．攻撃者は Algorithm 2 に示すように下記の手順を実行することで，ダミーの特定が可能である．

(1) 攻撃者はアプリケーションの初期状態のコピーを作成する．

(2) アプリケーションを実行し，各メモリアクセスで shelter(もしくはバッファ) に移動するブロックと出力を記録する．

(3) 任意のアクセスに対して shelter(もしくはバッファ) に移動するブロックに変更を加える．

(4) 加えた変更が途中結果もしくは出力に影響を与えるかどうかを確認する．出力に影響がなかった場合，そのブロックはダミーデータとして記録する．影響があった場合，どの処理で利用されたかを照会し，真のデータとして記録する．

(5) アプリケーションを初期状態に戻し，手順 3 と手順 4 を繰り返し実行することで，どのブロックがどの処理で利用されているかを特定する．

あるいは，手順 3 で shelter(もしくはバッファ) に移動するブロックの代わりに初期状態に対してランダムに選択したブロックを変更することも可能である．

4.2 能動的な攻撃者に対する square root solution の安全性評価

square root の動作はアクセスしようとしているデータが shelter $\mathcal{S}$ 内に格納されているかどうかによって場合分けが可能であるため、それぞれに対して攻撃者が shelter にコピーされるブロックがダミーであるかどうかを正しく推定できる確率を導出する．

(1) $x \in S$

アクセスしようとしているデータ x が shelter に格納され

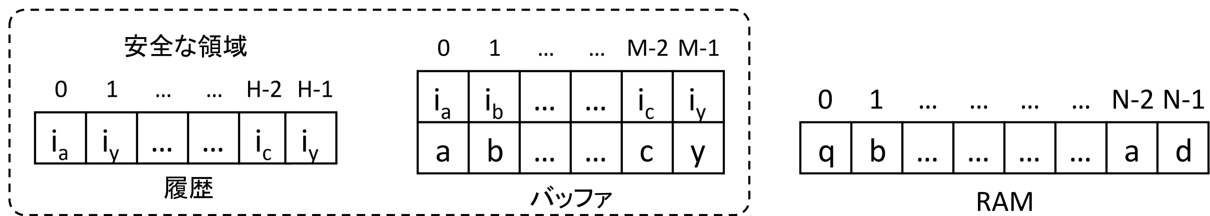

図 1 アクセスパターン保護手法の全体図
Fig. 1 Overview of Access Pattern Protection

Algorithm 2 攻撃手法

1: アプリケーションの初期状態のコピーを作成
2: 各メモリアクセスで shelter(もしくはバッファ) に移動するブロックと出力を記録
3: データに変更を加えない場合の出力を記録
4: **while** RAM 上にマークされていないデータが存在 **do**
5: 任意のアクセスで選択されたブロックを改変し，出力への影響を確認
6: **if** 出力が変化 **then**
7: 選択したブロックを真のデータとしてマーク
8: **else**
9: 選択したブロックをダミーとしてマーク
10: **end if**
11: **end while**

ている場合，shelter にコピーされるデータはダミーの位置から選択される．アクセスしようとしているデータ x が shelter に格納されている確率を p_S とすると，shelter にコピーされるデータがダミーである確率は

$$\frac{\sqrt{n}p_S}{n+\sqrt{n}}$$

となる．

(2) $x \notin S$
アクセスしようとしているデータ x が shelter に格納されていない場合，x が shelter にコピーされるため，ダミーである可能性はない．

(1)$x \in S$，(2)$x \notin S$ をまとめると，攻撃者がダミーを特定可能な確率は

$$\frac{\sqrt{n}p_S}{n+\sqrt{n}}$$

で表すことができる．$0 \leq p_S \leq 1$ であるため、攻撃者がダミーを特定できる確率は高々 $\sqrt{n}/(n+\sqrt{n})$ である．これはアクセスのパターンに関係なく RAM 上のデータをランダムに選択した時にそれがダミーである確率と等しく，攻撃者が優位性を得ないことを示す．

4.3 能動的な攻撃者に対する筆者らの保護方式の安全性評価

各アクセスでバッファに移動するデータのうち，少なくとも一つはダミーデータであることを利用し，ダミーデータの特定を行うことを検討する．メモリアクセスパターン保護方式の動作は i) $x \in M$, ii) $x \notin M, x \in H$, iii) $x \notin M$, $x \notin H$ の 3 つに場合分け可能であるため，それぞれに対してダミーを特定可能な確率を導出する．

(1) $x \in M$
この場合，必要なデータはバッファに存在するため，バッファに移動するブロックはどちらもダミーである．ただし，攻撃者はどのデータに対してアクセスしようとしているか，そのデータがバッファに格納されているかどうかを知ることができないため，どちらかをダミーとしてマークするとすると，攻撃者は確率 1 でダミーを特定可能である．$x \in M$ が発生する確率を p_M とすると，攻撃者がダミーを特定できる確率は p_M となる．

(2) $x \notin M, x \in H$
この場合，履歴バッファから選択されたブロックが必要なデータで，もう一方のブロックはランダムに選択される．したがって，攻撃者は 2 つのブロックのうち，いずれかをダミーとしてマークすることで確率 1/2 でダミーを特定可能である．$x \in H$ が発生する確率を p_M とした時，$x \notin M$, $x \in H$ が発生する確率は，$(1-p_M)p_H$ となる．したがって、攻撃者がダミーを特定できる確率は $(1-p_M)p_H/2$ となる．

(3) $x \notin M, x \notin H$
この場合，履歴バッファから選択されたブロックはランダムに選択され，もう一方のブロックは真のデータ (x) である．したがって，攻撃者は 2 つのブロックのうち，いずれかをダミーとしてマークすることで確率 1/2 でダミーを特定可能である．$x \notin M$, $x \notin H$ が発生する確率は，$(1-p_M)(1-p_H)$ となる．したがって、攻撃者がダミーを特定できる確率は $(1-p_M)(1-p_H)/2$ となる．

RAM からランダムに選択したブロックがダミーブロックである確率は $d/(n+d)$ であるため，(1) $x \in M$，(2) $x \notin M, x \in H$，(3) $x \notin M, x \notin H$ の確率をまとめると，攻撃者がダミーを特定可能な確率は

$$\frac{1+p_M}{2} \cdot \frac{d}{n+d}$$

で表すことができる．$0 \leq p_M \leq 1$ であるため、攻撃者がダミーを特定できる確率は高々 $d/(n+d)$ である．これはアクセスのパターンに関係なく RAM 上のデータをランダムに選択した時にそれがダミーである確率と等しく，メモ

© 2017 Information Processing Society of Japan

リアクセスパターン保護方式の特徴である「バッファに移動するブロックのうち，少なくとも一つはランダムに選択されたブロックである」ことを利用した場合でも，攻撃者が優位性を得ないことを示す．

5. 対策手法

本節ではメモリ上のデータを改変可能な攻撃者に対する対策手法とその効果について述べる．

5.1 多重化

複数のメモリ保護を並列で実行し，それぞれでデータが一致するかどうかを確認する．データが一致しない場合，攻撃者がデータを改変した可能性が高いので，処理を中止することでダミーデータを特定されることを防ぐことが可能である．また，３つ以上のメモリ保護を並列化し，データが一致しない場合，多数決によってデータを確定し，攻撃者が加えた変更を元に戻すことが可能である．しかし，いずれの場合も攻撃者が多重化されたすべてのデータを同時に変更することで，検出されることなくデータの変更が可能である．

5.2 誤り検出

全データに対してチェックサムを付与することで攻撃者によるデータの改変を検出できる．ただし，攻撃者はどこにチェックサムが格納されているかを特定できる可能性があるため，改変したデータに一致するようにチェックサムを改変できる場合，十分な効果が得られない．従って，鍵付きハッシュを使い，鍵を攻撃者がアクセスできない領域に格納することが必要である．例えば，チェックサムを

$$H(data||index||key)$$

として計算し，key のみを安全な領域に格納し，チェックサムはデータとセットで管理をする．データにアクセスする際にはデータとチェックサムをバッファに移動し，その後チェックサムを比較する．チェックサムはダミー，真のデータともに計算を行うため，攻撃者が変更したブロックがダミーであった場合もこれを検出し，プログラムを終了させることが可能であり，定義 1 を満たすことが可能である．

5.3 誤り伝搬

あるデータに対して攻撃者が変更を加えると，その変更が複数のデータに伝搬するような構成とすることで，攻撃者が変更したデータがダミーであってもプログラムの出力に影響を与えるような構成が可能である．難読化を用いることで，このような構成を実現する可能である [3]．難読化を利用する場合，まず複数のデータを一つにまとめたセットを構成する．その上で，同一セット内の各ブロックがそれぞれに依存するような変換を加える．これによって，一つのデータにアクセスする場合にも，そのデータが属するセット内の全データに対してアクセスを行い，元のデータを復元する必要が生じるが，セット内のあるデータが攻撃者によって変更された場合，その変更がほかのセットにも影響を及ぼす．各セットに少なくとも一つは真のデータを含めるような構成にすることで，変更したデータがダミーであってもプログラムの出力に影響を与えるためダミーの特定が困難になる．これによって定義 1 を満たすことが可能である．難読化の詳細は A.1 に示す．

6. まとめ

本稿では，メモリアクセスパターン保護方式が能動的な攻撃者に対してどの程度安全性を持つかの評価を実施した．元のデータ数を n，追加するダミーの数を d としたとき，攻撃者がランダムにデータを選択した場合，それがダミーである確率は $d/(n+d)$ である．Square root 方式の ORAM と筆者らのメモリアクセスパターン保護方式はメインメモリから安全な領域にデータを移動したのち，アクセスを行う．必要なデータが安全な領域にすでに格納されている場合はランダムに選択されたデータが移動するため，これを活用することで効率的にダミーを特定可能かどうかを評価した．その結果，いずれの方式も $n+d$ 個のデータからランダムに選択する方式のほうが効率的であることを示した．さらに，能動的な攻撃者がデータを改変した場合にそれを検出もしくは元のデータに復元する対策手法を提案した．今後の課題として，Path ORAM や他の ORAM 方式について同様の評価を実施することがあげられる．

参考文献

[1] Ajtai, M.: Oblivious RAMs without cryptographic assumptions, *STOC* (Schulman, L. J., ed.), ACM, pp. 181–190 (2010).

[2] Damgård, I., Meldgaard, S. and Nielsen, J. B.: Perfectly Secure Oblivious RAM without Random Oracles, *TCC* (Ishai, Y., ed.), LNCS, Vol. 6597, Springer, pp. 144–163 (2011).

[3] Fukushima, K., Kiyomoto, S., Tanaka, T. and Sakurai, K.: Analysis of Program Obfuscation Schemes with Variable Encoding Technique, *IEICE Transactions*, Vol. 91-A, No. 1, pp. 316–329 (2008).

[4] Goldreich, O.: Towards a Theory of Software Protection and Simulation by Oblivious RAMs, *STOC* (Aho, A. V., ed.), ACM, pp. 182–194 (1987).

[5] Goldreich, O. and Ostrovsky, R.: Software Protection and Simulation on Oblivious RAMs, *J. ACM*, Vol. 43, No. 3, pp. 431–473 (1996).

[6] Goodrich, M. T., Mitzenmacher, M., Ohrimenko, O. and Tamassia, R.: Oblivious RAM simulation with Efficient Worst-Case Access Overhead, *CCSW* (Cachin, C. and Ristenpart, T., eds.), ACM, pp. 95–100 (2011).

[7] Goodrich, M. T., Mitzenmacher, M., Ohrimenko, O. and Tamassia, R.: Practical Oblivious Storage, *CODASPY* (Bertino, E. and Sandhu, R. S., eds.), ACM, pp. 13–24

© 2017 Information Processing Society of Japan

(2012).

[8] Goodrich, M. T., Mitzenmacher, M., Ohrimenko, O. and Tamassia, R.: Privacy-Preserving Group Data Access via Stateless Oblivious RAM Simulation, *SODA* (Rabani, Y., ed.), SIAM, pp. 157–167 (2012).

[9] Kiyomoto, S., Rein, A., Nakano, Y., Rudolph, C. and Miyake, Y.: LMM - A Common Component for Software License Management on Cloud, *SECRYPT 2013* (Samarati, P., ed.), SciTePress, pp. 284–295 (2013).

[10] Kushilevitz, E., Lu, S. and Ostrovsky, R.: On the (In)security of Hash-based Oblivious RAM and a New Balancing Scheme, *SODA* (Randall, D., ed.), SIAM, pp. 143–156 (2012).

[11] Lorch, J. R., Mickens, J. W., Parno, B., Raykova, M. and Schiffman, J.: Toward Practical Private Access to Data Centers via Parallel ORAM, *IACR ePrint*, Vol. 2012, p. 133 (2012).

[12] Nakano, Y., Cid, C., Kiyomoto, S. and Miyake, Y.: Memory Access Pattern Protection for Resource-Constrained Devices, *CARDIS* (Mangard, S., ed.), LNCS, Vol. 7771, Springer, pp. 188–202 (2012).

[13] Pinkas, B. and Reinman, T.: Oblivious RAM Revisited, *CRYPTO* (Rabin, T., ed.), LNCS, Vol. 6223, Springer, pp. 502–519 (2010).

[14] Ren, L., Fletcher, C. W., Yu, X., van Dijk, M. and Devadas, S.: Integrity verification for path Oblivious-RAM, *HPEC*, IEEE, pp. 1–6 (2013).

[15] Ren, L., Yu, X., Fletcher, C. W., van Dijk, M. and Devadas, S.: Design space exploration and optimization of path oblivious RAM in secure processors, *ISCA* (Mendelson, A., ed.), ACM, pp. 571–582 (2013).

[16] Shi, E., Chan, T.-H. H., Stefanov, E. and Li, M.: Oblivious RAM with $O((\log N)^3)$ Worst-Case Cost, *ASIACRYPT* (Lee, D. H. and Wang, X., eds.), LNCS, Vol. 7073, Springer, pp. 197–214 (2011).

[17] Stefanov, E., van Dijk, M., Shi, E., Fletcher, C. W., Ren, L., Yu, X. and Devadas, S.: Path ORAM: an extremely simple oblivious RAM protocol, *ACM Conference on Computer and Communications Security* (Sadeghi, A.-R., Gligor, V. D. and Yung, M., eds.), ACM, pp. 299–310 (2013).

[18] Williams, P. and Sion, R.: Usable PIR, *NDSS*, The Internet Society (2008).

[19] Williams, P. and Sion, R.: Single round access privacy on outsourced storage, *ACM Conference on Computer and Communications Security* (Yu, T., Danezis, G. and Gligor, V. D., eds.), ACM, pp. 293–304 (2012).

[20] Williams, P., Sion, R. and Tomescu, A.: PrivateFS: a parallel oblivious file system, *ACM Conference on Computer and Communications Security* (Yu, T., Danezis, G. and Gligor, V. D., eds.), ACM, pp. 977–988 (2012).

付　　録

A.1 難読化用行列

データを変数 $\boldsymbol{v}$ として実装した場合，次に示すように難読化することで $\boldsymbol{x}$ へ変換される．

$$\boldsymbol{x} = M \times \begin{pmatrix} \boldsymbol{v} \\ d \end{pmatrix} \oplus \boldsymbol{y},$$

ここで，d は 1 ブロックの変化が $\boldsymbol{x}$ 全体に影響するようにするための乱数である．本実装では，16×9 の行列を選択する．これによって，8 ブロックが 16 ブロックへと変換される．使用した行列 M とベクトル $\boldsymbol{y}$ は次のとおりである．

$$M = \begin{pmatrix} 111011111 \\ 101111111 \\ 110001001 \\ 011011101 \\ 111000101 \\ 101011001 \\ 111100101 \\ 111110111 \\ 000011011 \\ 000010001 \\ 101100101 \\ 101011011 \\ 000110001 \\ 111110101 \\ 000000111 \\ 100101001 \end{pmatrix}, \boldsymbol{y} = \begin{pmatrix} \texttt{0xf8604474} \\ \texttt{0x04656b7a} \\ \texttt{0xaf29b326} \\ \texttt{0x13a4a306} \\ \texttt{0x4b655263} \\ \texttt{0xe624d88a} \\ \texttt{0xada199e2} \\ \texttt{0xce2d6a2c} \\ \texttt{0x246214f0} \\ \texttt{0xbd6d9441} \\ \texttt{0x962df3b9} \\ \texttt{0x57572c62} \\ \texttt{0x65ecb628} \\ \texttt{0x085b5ffa} \\ \texttt{0x8b3d4784} \\ \texttt{0xdaea6eb2} \end{pmatrix}$$

ここで，$\boldsymbol{y}$ は 16 進数で表されている．8 ブロックごとに難読化を適用し，十分なサイズの安全な領域を構築する．

© 2017 Information Processing Society of Japan