オープン環境におけるプログラム保護技術の研究開発

Study of Program Protection Mechanism for Open Platform Architectures

研究代表者
田中俊昭 株式会社 KDDI 研究所
Toshiaki Tanaka KDDI R&D Laboratories Inc.

研究分担者
三宅優[†] 清本晋作[†] 仲野有登[†]
Yutaka Miyake[†] Shinsaku Kiyomoto[†] Yuto Nakano[†]
[†]株式会社 KDDI 研究所
[†]KDDI R&D Laboratories Inc.

研究期間 平成 24 年度

概要
クラウドやオープンプラットフォームなどの安全性の確保が従来のシステムに比べて困難な環境で実装されるプログラムを保護するための技術を確立する。これに加えて、プログラム保護技術を応用してソフトウェアでトラストアンカーを構築することで、サーバに保存されたデータの管理状況をユーザがセキュアに検証できる技術やサーバ上で実行するプログラムを保護する技術を検証する。これらの実現のために、従来とは異なる基礎技術を使用し、性能低下を最小限とするセキュリティ対策を提案する。

１．まえがき

従来、事業者から提供されている実行環境は信頼できるという前提のもとでプロトコルやソフトウェアに関するセキュリティ技術の検討が進んできた。しかし、パブリッククラウドなど、オープン環境では、この前提が必ずしも満たされるわけではない。したがって、オープンな環境においても安全な実行環境を構築するための技術として様々なものが検討されているが、実用化のためには解決すべき課題が数多く残されている。そこで、本研究開発では、オープン環境における安全なプログラム保護技術を研究開発することで、安全性が低いと考えられているオープンな環境においてもサービスを安全に実行し、その実行状況をセキュアに保護・検証・監視できる実用的な方式の実現に寄与する。

２．研究開発内容及び成果

オープン環境におけるプログラム保護技術の研究開発は、図1に示すように2つの研究課題から構成される。メモリアクセスパターン保護技術の検討により確立された安全なプログラム実行を実現するメモリ保護技術は、プログラムがアクセスするデータをサイドチャネル攻撃などから保護する。また安全な実行プログラムの構築技術においては、ライセンス情報を安全かつ柔軟に運用するライセンス管理モジュールを構築し、アプリケーションが様々な環境で安全に利用できるようにする。ライセンス管理モジュール内において、メモリアクセスパターン保護技術はライセンス情報などの重要なデータにアクセスする際に利用され、安全な実行プログラムの構築に寄与する。

以下、各課題について説明を行う。

ア）安全なプログラム実行を実現するメモリ保護技術では、信頼できない環境において実行されるプログラムに関して、そのプログラムが行うメモリに対するアクセスのパターンを攻撃者から保護する技術の検討を行った。難読化によって保護された領域と保護されていないメインメモリ間でデータをやり取りする場合を考える。複数回アクセスされるデータの場合、二つの領域で何度もやり取りされることになるので、（そのデータが何であるかの特定は不可能であるが）あるデータが複数回アクセスされていることを推測できる。そこで本研究開発では、アクセスの履歴を保存することで、故意に二つの領域でデータがやり取りされるような方式を検討した。さらに、データの管理手法の効率化や、使用メモリの削減など、実用化に向けたパフォーマンスの向上に関しても検討を行い、検討した手法を実装した。また、課題イで実装するライセンス管理モジュールに組み込み、その妥当性を評価した。

イ）安全な実行プログラムの構築技術では、プログラムのライセンス管理を適切かつ柔軟に行い、プログラムを安全に実行する技術の研究開発を行った。本技術を用いることで、アプリケーションの起動時に、ライセンス情報の正当性を確認し、アプリケーションの不正利用を防止することが可能である。一方で、クラウド環境では複数のサーバの中からその時点で最適なサーバを選択し、その上で実行するなど、従来とは異なるライセンス管理の形態が必要となりつつある。そこで、本研究開発では、ライセンス管理モジュールを実装し、オンライン／オフラインでライセンス情報の確認するモデルを検討し、モジュールの有効性を確認した。加えて、異なるクラウドサーバ間で柔軟なアプリケーションの実行を実現するため、ライセンス管理モジュールを他のサーバに移譲して、認証を受ける機能を実現した。

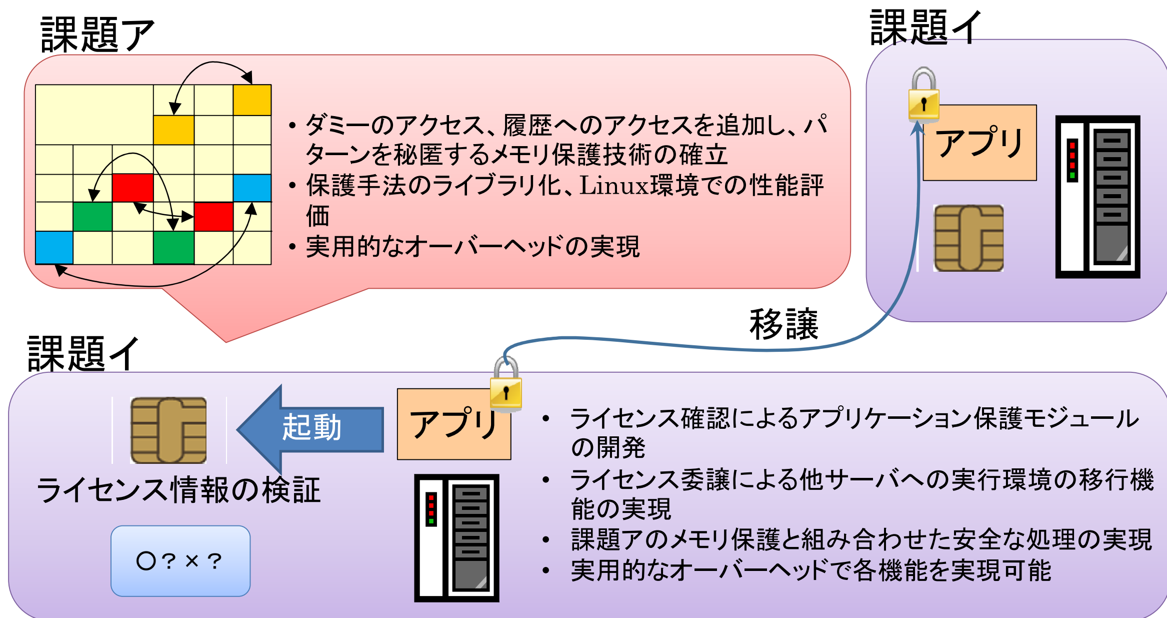
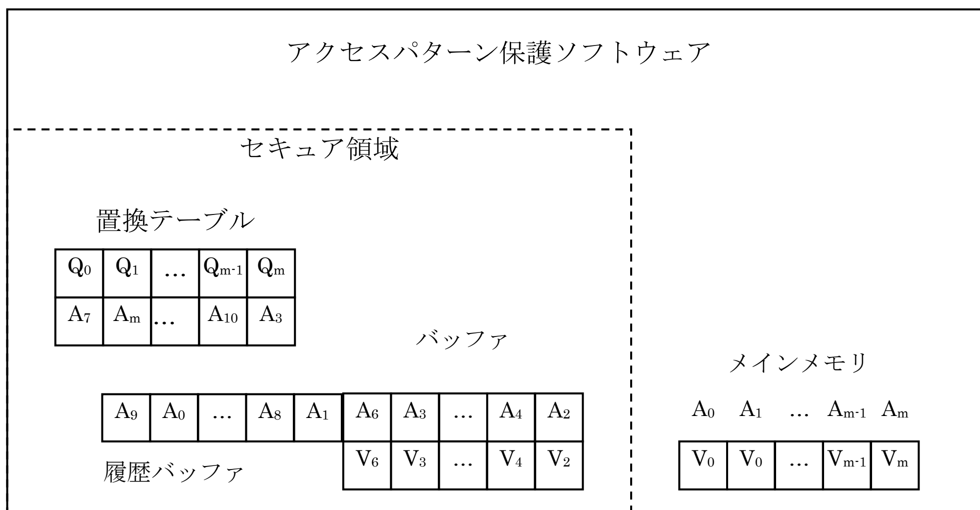


図1　オープン環境におけるプログラム保護技術



V はデータ、A は V のアドレス、Q は擬似アドレスを示す。

図 2　メモリアクセスパターン保護ライブラリ

ア）安全なプログラム実行を実現するメモリ保護技術

課題ア）では、プログラムがメモリアクセスを行っている最中に攻撃者がメモリを監視していたとしても、どのようなパターンでデータに対してアクセスしているのかを秘匿する技術を確立した。これによって、攻撃者が重要なデータを入手することを防ぐことができる。提案した手法に関して、理論的なオーバーヘッドの評価を行った後、実装を行い、実機でのパフォーマンス評価を行った。その結果、理論的なオーバーヘッドは5倍に抑えられており、目標を達成した。また、実機でのパフォーマンス評価において、メモリ保護技術をPC上に実装し、1MBのデータをロードする際に必要となるオーバーヘッドを計測したところ、約130[ms]から約470[ms]に抑えられており、高速に動作可能なメモリ保護技術を確立した。

図2に安全なプログラム実行を実現するメモリ保護技術のためのメモリアクセスパターンソフトウェアの概念図を示す。本ソフトウェアは共通ライブラリとして提供され、外部のソフトウェア内の保護対象の処理は本ライブラリを通じてメモリにアクセスを行う。外部のプログラムが使用するデータは、バッファおよび履歴テーブルのデータに応じて、メインメモリからコピーされる。

まず、アクセスパターン保護手法の詳細について、図３に示した例を用いて述べる。図３では、初期状態から順にプログラムがデータにアクセスしていく場合の例を示している。本手法では初期化処理として、置換テーブルによってプログラムが使用するデータがランダムに並び替えられる。次に、バッファのサイズ分だけランダムにデータを選択し、バッファに移動させる。その後、以下の手順でアクセスが行われる。

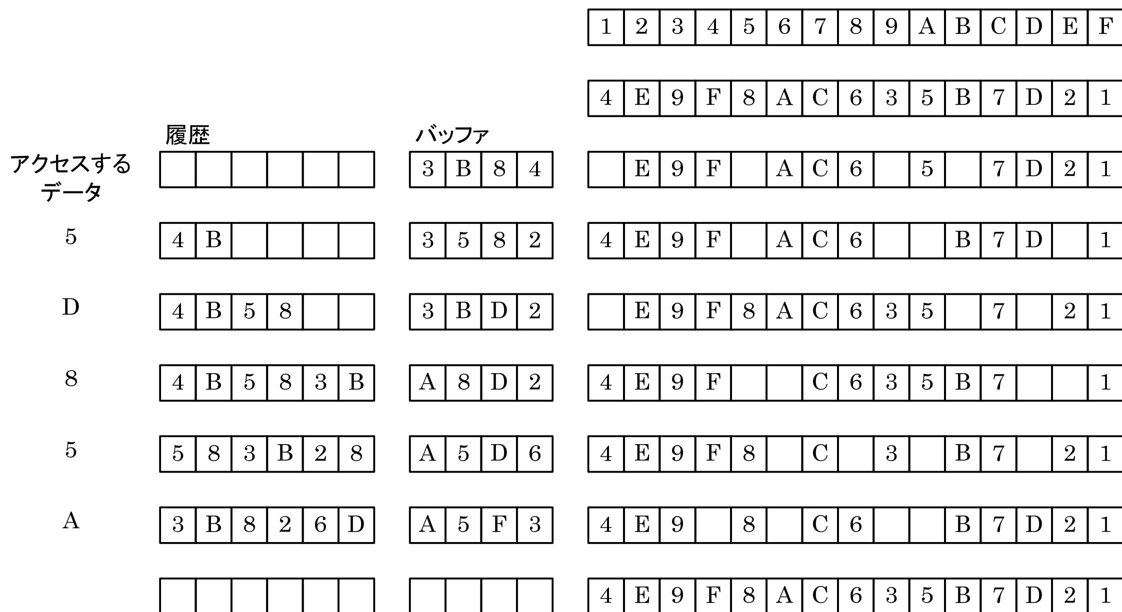


図 3　メモリアクセスパターン動作

1. データブロック 5 にアクセスする場合、5 はバッファに含まれないため、5 ともう一つランダムに選択されたデータ(この場合は 2)をバッファに移動する。4 と B がバッファから移動したのでこの二つを履歴に保存し、プログラムは 5 にアクセスする。この時、ほかのデータ(この場合は 2、3、8)に対してもアクセスを行い、5 にアクセスしていることを秘匿する必要がある。
2. データブロック D にアクセスする場合、D はバッファに含まれないため、D をバッファに移動する。もう一つのデータは履歴より選択する。この場合はブロック B を選択する。代わりに 5 と 8 がバッファからメモリに移動し、履歴に保存される。プログラムは D にアクセスする。
3. データブロック 8 にアクセスする場合、8 は履歴に含まれているため、もう一つのデータは履歴に含まれていないデータをランダムに選択し(この場合は A)、バッファに移動する。代わりに 3 と B が書き出され、履歴に保存される。プログラムは 8 にアクセスする。
4. プログラムが 5 にアクセスする場合、5 は履歴に含まれているため、もう一つのデータは履歴に含まれていないデータをランダムに選択し(この場合は 6)、ブロックに移動する。代わりに 2 と 8 が書き出され、履歴に保存される。プログラムは 5 にアクセスする。
5. プログラムが A にアクセスする場合、A はバッファに保存されているため、ランダムなデータと履歴に保存されているデータ(この場合、3 と F)をバッファに移動し、6 と D が書き出される。プログラムは A にアクセスする。
6. すべての処理が完了するとバッファ内の必要なデータをメモリに書き出し、プログラムは終了する。

以上のように、履歴を保存し、履歴とアクセスしようとしているデータに応じて、メモリとバッファの間で常に二つのデータブロックをやり取りする。これによって、あるデータに対する複数回のアクセスが、プログラムによって実行されているアクセスであるのか、そのデータが履歴に保存されているからアクセスされているのかが判別困難となり、アクセスパターンを秘匿することが可能である。

安全性評価および効率性とのトレードオフ

本節では、提案したメモリアクセスパターン保護手法の安全性を理論的に評価する。まず、アクセスパターン保護方式に対して、δ長アクセスとδ長ϵ安全性を次のように定義する。

定義　δ長アクセス
データへのアクセスに対して、あるデータ a に対するアクセスが t 番目のアクセスと$t+\delta$番目のアクセスである時、この二つのアクセスを a に対するδ長アクセスと呼ぶ。

定義　δ長ϵ安全性
アクセスパターン A(y)において、アクセスの間隔が$d \leq \delta$であるような任意の二つのアクセスに対して、攻撃者が二つのアクセスを特定できる確率が高々ϵであるとき、アクセス保護手法はδ長ϵ安全性を満たすという。

提案手法の初期化において、n 個の元データに対して合計がkn（$k \geq 1$を満たす定数）となるようにダミーのデータを追加し、さらに法則$\mathcal{E}$に従ってメモリ上のデータを並び替える。バッファのサイズをmとし、メモリのサイズをℓとする。ただし、$m \ll \ell$を満たす。また、$kn = m + \ell$が成り立つ。さらに履歴を保存しておく領域を$\mathcal{H}$と表し、$\mathcal{H}$のサイズをℓ_hとする。データブロック a のアドレスはi_aとして$\mathcal{H}$に保管される。

データaに対して繰り返しアクセスが行われる際のアクセスパターン保護の安全性を評価する。データaに対するアクセスは次の 3 パターンに分類され、それぞれの場合においてどのデータがバッファにコピーされるかは次の通りである。

1. $a \in \mathcal{M}$でaがアクセスされる場合、(r, p)をメモリ

からバッファにコピーする

2. $a \notin \mathcal{M}$かつ$a \in \mathcal{H}$でaがアクセスされる場合、(r,a)をメモリからバッファにコピーする
3. $a \notin \mathcal{M}$かつ$a \notin \mathcal{H}$でaがアクセスされる場合、(r,a)をメモリからバッファにコピーする

ここで、r はメモリ中からランダムに選択されたデータを表し、p は履歴中からランダムに選択されたデータを表す。時刻 t におけるアクセスをX_tで表し、時刻 t におけるアクセスが a に対するアクセスである場合は$X_t = a$と書く。

補題 1
$X_t = a$とする。ここからδステップ後にデータaがバッファおよび履歴に残っている確率はそれぞれ、$p_M = \Pr[a \in M] \geq \left(\frac{m-2}{m}\right)^{\delta}$，$p_H = \Pr[a \in H] \geq \left(\frac{\ell_h}{\ell_h+2}\right)^{\delta}$で表される。

証明
$\Pr[a \in M] \geq \left(\frac{m-2}{m}\right)^{\delta}$において、右辺は 1 回あたり 2 つのデータを入れ替える操作をδ回行った際にデータがバッファに残っている確率であり、データが 1 度バッファから出た後に再度バッファにコピーされることを考慮し不等号を含めて表している。$\Pr[a \in H] \geq \left(\frac{\ell_h}{\ell_h+2}\right)^{\delta}$も同様に計算されるが、履歴は$\ell_h + 2$から 2 つを選択されるため、1 回あたり$\left(\frac{\ell_h}{\ell_h+2}\right)$で表される。

補題 2
上記で検討した 3 つの場合分けにおいて、$X_t = \mathrm{X}_{t+\delta} = a$が満たされる確率はそれぞれ次のようになる。1 の場合:p_M、2 の場合$(1-p_M)$、3 の場合$(1-p_M)(1-p_H)$

証明
補題 1 の証明および場合分けの条件から明らかである。

定理 1
提案方式はδ長ϵ安全性を満たす。ただし、ϵは次のように与えられる。

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

証明
攻撃者は$X_i = a_i$を観測できるとする。また、$X_t = \mathrm{X}_{t+\delta} = a$であるとする。この時、$\Pr[X_t = \mathrm{a}, \mathrm{X}_{t+\delta} = a]$を求めればよい。時刻$t$と時刻$\mathrm{t} = \delta$のアクセスは独立であると仮定し、データに対するアクセスは一様に分布していると仮定すれば$\Pr[X_t = a] \leq 1/(m-2)$であるので、補題 1, 2 から

$$\begin{aligned}\Pr[X_t = \mathrm{a}, \mathrm{X}_{t+\delta} = a] &= \Pr[X_t = a] \times \Pr[\mathrm{X}_{t+\delta} = a] \\ &\leq \frac{1}{m-2}(\Pr[X_{t+\delta} = a \mid \mathrm{case1}] \\ &\times \Pr[\mathrm{case1}] + \Pr[X_{t+\delta} = a \mid \mathrm{case2}] \\ &\times \Pr[\mathrm{case2}] + \Pr[X_{t+\delta} = a \mid \mathrm{case3}] \\ &\times \Pr[\mathrm{case3}]) \\ &\leq \frac{p_M}{(m-2)^2} + \frac{(1-p_M)p_H}{(m-2)^2} \\ &+ \frac{(1-p_M)(1-p_H)}{2(m-2)}\end{aligned}$$

を得る。したがって

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

である。

定理 1 で示したように、安全性を決定するパラメータはm, ℓ_h, δの 3 つである。このうち、δはプログラムによって決定される値であるため、メモリ保護の実装において、開発者が自由に設定できるパラメータはm, ℓ_hの二つである。提案方式の説明で述べたように、アクセスしようとするデータがバッファもしくは履歴に含まれているかどうかを判定するためには、これらの領域をスキャンする必要があり、m, ℓ_hが大きくなるにつれてスキャンにかかる時間も増加する。したがって、より高い安全性を実現するためにm, ℓ_hを大きくすればするほど効率性が低下する。したがって、利用する環境やプログラムに応じた安全性と効率性を実現するため、最適なパラメータ選択の指針として定理 1 が活用可能である。

メモリアクセスパターン保護プログラムの効率化手法

本手法をソフトウェアのみで実装した場合、次のような課題がある。

1. ソフトウェアのみでの実現のため、バッファはメモリ上に展開され、攻撃者もバッファにアクセス可能であると想定される。その場合、プログラムがデータにアクセスする際、バッファ内の全データに対してアクセスを行わなければ、どのデータが真のアクセスであるかの情報が攻撃者に漏えいしてしまう。そこで、これを防ぐためにはバッファ内の他のデータに対してもダミーのアクセスを行い、真のアクセスを秘匿する必要がある。しかし、これが大きなオーバーヘッドとなっている。
2. バッファ内には常に最新のデータが一つだけ保管されるように管理しなければならない。そのため、バッファとメモリ間でデータの交換を行う際には、交換しようとしているデータがすでにバッファに存在しているかどうかをスキャンによって確認する必要があり、これが大きなオーバーヘッドを引き起こす。
3. データの交換を行うため、全データブロックが同じサイズである必要がある。このため、ブロックサイズよりも小さなデータを多く扱うプログラムが本機能を利用する場合、実際よりも多くのメモリを消費することが懸念される。

これらの課題に対して、本節では、メモリアクセスパターン保護プログラムの効率化について、難読化によるセキュア領域の確保、フラグを用いたデータ管理、使用メモリの効率化の 3 点の改良手法を述べる。難読化によるセキュア領域の確保では、バッファ全体のデータをスキャンする場合に比べて、スキャンする領域を小さくすることが可能であり、パフォーマンスの向上が実現できる。フラグによるデータの管理では、バッファサイズが大きくなるにつれて、そのスキャンにかかる時間が膨大になるため、フラグを用いて、データがバッファに保存されているかどうかを判定する。使用メモリの効率化では、一つのブロックに複数のデータを保存し、使用メモリを削減する。以下に詳細を述べる。

セキュア領域の確保

図 4 にバッファに対する難読化の適用を示す。バッファを細かいサイズに区切り、それぞれを独立に難読化することで可変サイズのバッファに対して難読化を適用することが可能となり、より柔軟な構成が実現できる。

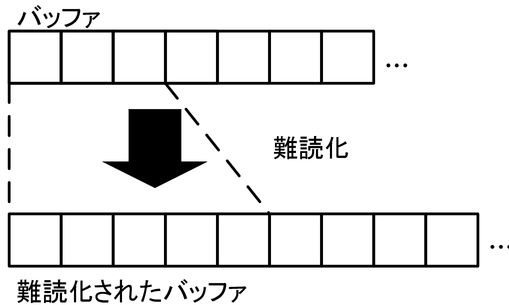


図 4　バッファに対する難読化

具体的な難読化手法について、簡単な例を用いて示す。v_1、 v_2、 v_3 を元の変数とし、これらに難読化を適用した結果を x_1、 x_2、 x_3、 x_4、 x_5 とすれば、変換は

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix} \times \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ d \end{pmatrix} \oplus \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{pmatrix}$$

と表すことができる。ここで、d は乱数を表す。式に表される通り、乱数 d とかけられる列は常に 1 になっており、これによって元の変数 v の変更が難読化後の変数全体に及ぶようになっている。難読化を適用することによって、1 つの変数を変更すれば、それが 5 つの変数に影響を及ぼすことになる。また、難読化するブロックサイズを状況に応じて変更することで、必要に応じて安全性と性能を選択できる。

フラグを用いたデータの管理

バッファ内に保管されているデータの管理することが重要であり、最新のデータが一つだけ保持されていなければならない。そのため、メインメモリからバッファにデータを移動する際に、選択したデータがバッファに保持されていないことを確認する必要がある。シンプルな解決策としてスキャンによるものがあるが、スキャンを使用した場合、多くのオーバーヘッドが生じる。そこで、メインメモリに保存されている全データブロックに 1 ビットの値を割り当て、ビットの 1, 0 によって各データがバッファに保持されているかどうかを判定する。これによって、バッファに保存されているデータの管理を効率的に行うことが可能である。

図 5　フラグによるデータの管理

図 5 に例を示す。今、バッファには 3、5、A、C、E が保持されており、メモリの対応する位置にフラグとして 1 を示す。それ以外のデータに関しては 0 が保持される。

使用メモリの効率化

各ブロックのサイズは一定であることが想定されている。したがって、大きなサイズのデータブロックに対応するために、各ブロックを CPU のワードサイズに合わせて 32 あるいは 64 ビットで構成する必要がある。しかし、プログラムが扱う変数はこれよりも小さい場合が多く、大きなブロックを使用すると必要となるメモリが増加するという問題がある。そこで、1 ブロックのサイズを 32 あるいは 64 ビットとし、1 つのブロックに複数の小さいデータを格納する。例えば、short int 型の場合、1 つのデータは 2 バイト（16 ビット）であるので、64 ビット環境では、1 つのブロックに 4 つの short int 型を格納することが可能となる。これによって、必要なメモリサイズが最大で 1/4 に削減可能となる。同時に、4 つの short int 型変数を 1 度に取得することが可能となるので、連続してアクセスするデータが同一のブロックに格納されている場合はアクセス速度の向上も実現できる。図 6 に 64 ビットブロックとした際の効率化例を示す。

メモリアクセスパターン保護プログラム評価結果

理論評価

バッファと履歴テーブルを一つのテーブルとして実装し、フラグを用いてデータを管理する。

あるデータに対してアクセスを行う場合、そのデータがバッファもしくは履歴テーブルに保管されているかどうかを 1 回のスキャンで判定し、スキャン結果に応じたデータをバッファとメインメモリ間で移動する。その後、プログラムがデータの読み出しと書き込みを行い、1 ステップの動作が完了する。したがって、必要な処理は、スキャン 1 回、データのスワップ 2 回、読み出し 1 回、書き込み 1 回の合計で 5 となり、オーバーヘッドは 5 倍となる。

図6　効率的なデータブロック利用

表2　計測結果

コピー処理の種類	平均処理時間(秒)	コピーする データサイズ	計測時の設定
バイト列からバイト列	9.20E-05	1024*1024バイト	memcpyを使用
バイト列から保護手法 （提案方式）	0.130443	1024*1024バイト	データ領域サイズ　：　10*1024*1024バイト バッファサイズ　：　100ブロック 履歴テーブルサイズ：　100ブロック
バイト列から保護手法 （既存方式）	1.0294	1024*1024バイト	ブロックサイズ　：　8 ブロック数　：　1024*1024/8 パーティション数　：　128 バックグラウンド排除率　：　1.0 ステップごとの最大処理数：　1

実装評価

表１に示す環境において、提案したメモリアクセスパターン保護手法を実装し、1024×1024 バイト(1MB)のデータをロードするために必要な時間を計測した。計測は 10 回行い、その平均を処理時間として評価する。計測結果は表２に示すとおりである。比較のため、memcpy を使用してデータを単純にコピーした場合の計測時間と、既存方式の処理時間も示す。これら二つについても同様に 10 回計測を行った平均時間を示す。

表１　評価環境

CPU	Core i7 3720QM
メモリ	16GB
OS	Ubuntu 12.10 64bit
コンパイラ	g++ 4.7.2
最適化レベル	O2

提案方式では、利用環境及び必要な安全性に応じて、バッファサイズと履歴テーブルを変更可能である。図７にバッファサイズと履歴テーブルのサイズを変更した際の処理時間の変化を示す。評価環境は表１に示したものと同一である。図７に示したとおり、処理時間はバッファ及び履歴テーブルのサイズに比例して増加しており、メモリアクセスパターン保護プログラムの実装が最適化されていることが分かる。

また、メモリアクセスパターン保護の関連技術などを調査した結果、現時点ではクラウド環境でセキュリティ上重要なデータを扱う処理が適用先として適していることが分かったため、課題イ）　安全な実行プログラムの構築技術の評価と合わせて行う。

安全なプログラム実行を実現するメモリ保護の実現のため、メモリアクセスパターン保護方式の検討及び改良を行った。また、検討した方式について、基本モジュールを作成し、性能評価を実施した。その結果、オーバーヘッドの理論値を目標値内に抑えつつ、既存の保護技術と比較して優れた処理性能を有することがわかり、実用的なメモリ保護技術を確立できた。

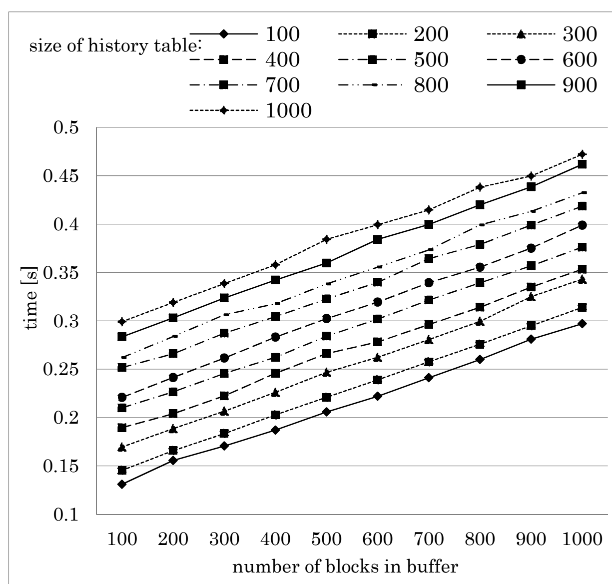


図７　バッファサイズおよび履歴テーブルと処理時間の関係

イ）安全な実行プログラムの構築技術

課題イ）では、ライセンス認証に成功した場合のみクラウド上のプログラムを実行可能とするシステムを構築した。安全な領域を確保し、その内部にライセンス情報を保管する。プログラム起動時にライセンス情報の検証を行い、正しい情報が確認できた場合にのみプログラムを実行することができる。また利用シナリオに即したライセンス管理モデルを提案し、各モデルにおける管理モジュールの動作を検討する。ライセンス検証は課題ア）の安全なプログラム実行を実現するメモリ保護技術と連携したシステムを構成し、安全にモジュールが実行されることを保証する。最後に、パフォーマンス評価を行った。その結果、十分小さいオーバーヘッドでライセンス管理モジュールが動作しており、安全な実行プログラム構築技術を確立することができた。

利用者はライセンス情報を購入し、それをクラウド上にアップロードする。クラウド上のアプリケーションを利用する際には、アプリケーションの起動時に、ライセンス情報の正当性が確認される。正当性が確認できた場合にのみ、アプリケーションを実行することが可能であり、これによって、アプリケーションの不正利用を防止することが可能である。クラウド環境では複数あるサーバ群から最適なサーバを選択し、実行するなど、従来とは異なる環境でアプリケーションを実行する環境が考えられる。そこで、オンライン／オフラインでのライセンス検証や他のサーバにライセンス情報を移譲するなどのモデルを検討し、各モデルにおけるモジュールの動作を検討する。

ライセンス管理モジュール(License Management Module、 LMM)

LMM の概要は、図 8 のとおりである。LMM は、LMM ライセンスファイルを管理する。LMM ライセンスファイルには、複数のアプリのライセンス情報が格納される。LMM の動作環境チェック機能により、他の環境にコピーしても実行できない。従って、LMM は各環境にそれぞれ実装されているものとする。

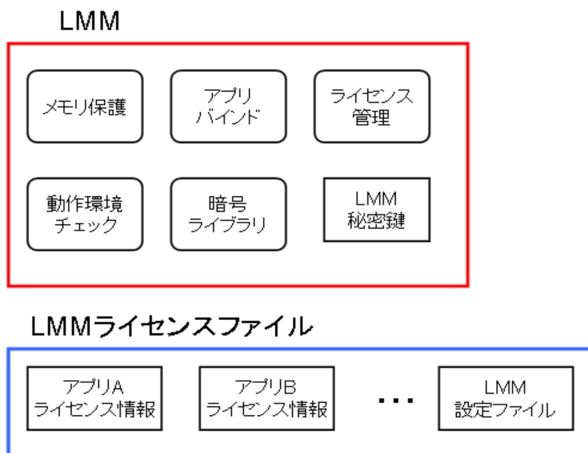


図 8　LMM の概要

ライセンス管理モデル

ライセンス管理モデルとして、一つのアプリケーションに対して、一つのライセンスが割り当てられており、ユーザによってライセンス情報が異なるという状況を想定する。また、正規のユーザであっても、二つ以上の環境で同じアプリケーションを同時に実行することはできないとする。この仮定のもと、4 つのモデルを提案し、それぞれの特徴および安全性について議論する。

モデル A

モデル A の構成を図 9 に示す。
モデル A は、オンライン方式であり、ライセンス情報はユーザ端末内の LMM によって管理される。プログラムはクラウド上で実行され、クラウドはパブリックなネットワークを通して、LMM にアクセスする。
利点：クラウド上にライセンス情報が保管されない
欠点：クラウドとユーザ端末間で頻繁に通信が発生する。

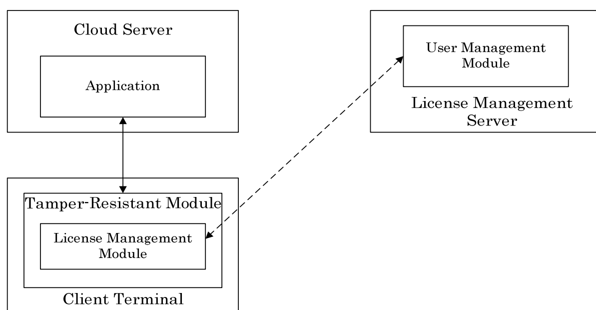


図 9　ライセンス管理モデル A

モデル B

モデル B の構成を図 10 に示す。
クラウド上とユーザ端末に LMM を配置し、プログラムの実行前に二つの LMM 間で移譲プロトコルを実行し、クラウド上にライセンス情報を移動させる。
利点：プログラム実行中にクラウドとユーザ端末間の通信が不要。
欠点：LMM を保存しておくクラウドを信頼するか、LMM を保護する必要がある

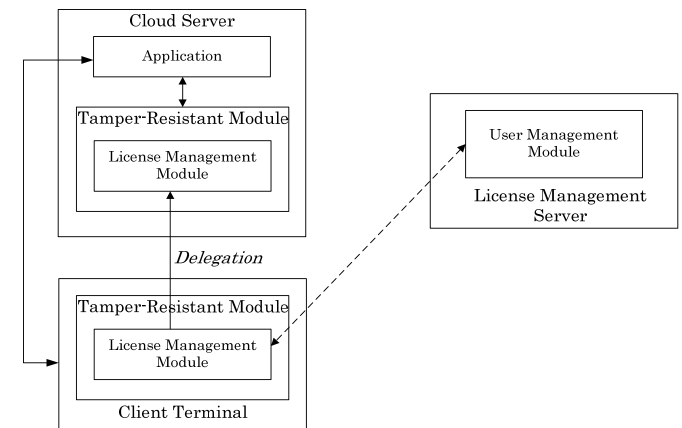


図10　ライセンス管理モデルB

モデル C

モデル C の構成を図 11 に示す。
モデル B の拡張であり、LMM がクラウド上で管理される。

利点：LMM を管理しているクラウドとの通信が不要
欠点：LMM を保存しておくクラウドを信頼するか、LMM を保護する必要がある

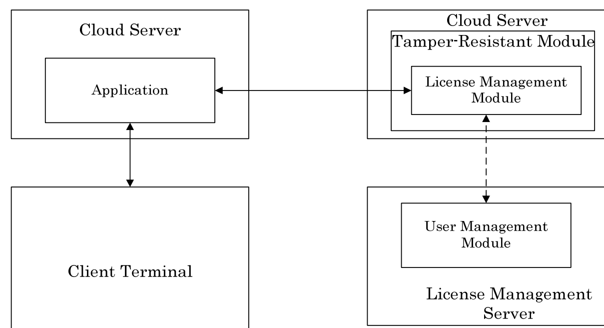


図11　ライセンス管理モデルC

モデル D

モデル D の構成を図 12 に示す。
モデル D はモデル B と C の組合せであり、クラウド上に保存されている LMM を別のクラウドに移譲する。
利点：ユーザ端末とクラウド環境との通信が不要である。
欠点：LMM を保存しておくクラウドを信頼するか、LMM を保護する必要がある

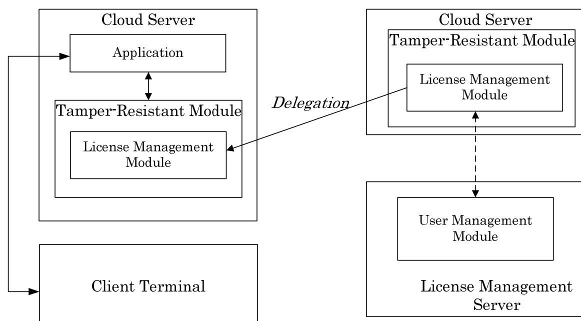


図 12　ライセンス管理モデル D

各モデルに関して、想定される脅威と、攻撃者及びターゲットを表 3 にまとめる。それぞれのモデルで、想定される脅威が異なることがわかる。LMM はこれらすべてのモデルに適用可能な教習モジュールとして設計した。また、今回の検討では、クラウド事業者が不正する可能性も考慮している。従って、LMM にはクラウド事業者からの攻撃に対して、安全性を確保する技術を具備することを求める。

表 3 各ライセンス管理モデルに対する脅威

想定される脅威		攻撃者	ターゲット	モデル A	B	C	D
ライセンスなしでの実行		ユーザ 事業者	プログラム(クラウド上)	✓	✓	✓	✓
ライセンスの不正利用		ユーザ 事業者	LMM		✓	✓	✓
ライセンスの漏洩	端末から	ユーザ	LMM	✓	✓		
	クラウドから	ユーザ 事業者	LMM		✓	✓	✓
通信に対する攻撃		ユーザ 事業者 第 3 者	通信データ	✓	✓	✓	✓
ライセンスの不正コピー	端末から	ユーザ	LMM	✓	✓		
	クラウドから	ユーザ 事業者	LMM		✓	✓	✓

アプリのライセンスチェック

アプリのライセンスチェックの手順を、表 4 にまとめる。アプリからの要求に対して、ライセンス情報を使って演算を行い、その結果をアプリに返信する。ライセンス情報が正しくない場合には、正しく演算できないため、ライセンスの有無を確認することができる。なお、ライセンス情報は、メモリに読み出される際に、前述のメモリアクセスパターン保護技術によって保護される。

表 4　ライセンスチェック時の処理

手順	対象	内容
1	アプリA	アプリID、(x1、x2、x3、x4)をLMMへ送信。
2	LMM	動作環境が正しいかの検証要求を行う。
3	LMM	クラウドシステムから環境情報パラメータを取得する。
4	LMM	環境パラメータのハッシュ値を計算し、LMM設定情報と比較する。 比較に失敗した場合は、エラーを返す。
5	LMM	アプリIDから、対応するアプリのライセンス情報を取得する。
6	LMM	LMM秘密鍵を読み込む。
7	LMM	LMM秘密鍵を用いて、アプリAのライセンス情報を復号する。
8	LMM	アプリIDに対応するライセンス情報と(x1、x2、x3、x4)を元に、(y1、y2、y3、y4)の値の計算を行う。 本計算には、難読化機能を利用する。
9	LMM	計算後の(y1、y2、y3、y4)の値をアプリケーションに送付する。
10	アプリA	LMMから受信した(y1、y2、y3、y4)の値のチェックを行う。

ライセンスの移譲

アプリのライセンスの移行では、まず、送信元の LMM に保管されているライセンス情報を LMM の共通鍵をつかって暗号化する。共通鍵が抜き取られるのを防ぐため、本暗号化はメモリ保護技術を用いて保護される。暗号化されたライセンス情報はライセンス管理を経由してライセンス移行アプリから送信先のライセンス移行アプリへと送信される。送信先ではライセンス管理を経由したのち復号され、ライセンス情報が保管される。復号処理も同様にメモリ保護を用いて保護される。

性能評価

ライセンス管理モジュールを PC 上およびクラウド上で実装し、各機能の処理時間を計測した。測定環境は表 5 に示すとおりである。また、この時の LMM のコードサイズは 314KB であった。表 6 に評価結果をまとめる。処理時間はメモリアクセス保護技術の成果を組み込んだ結果であり、ライセンス移譲に関しては、AKE プロトコルにかかる時間は含まれない。すべての処理において、メモリ保護機能を含めた処理時間は非常に小さく、LMM が実サービスにおいて有効であることを確認した。

安全な実行プログラムの構築技術について、ライセンス管理モジュールの利用モデルを 4 つ提案し、それぞれの方式検討及び改良を行った。また、課題ア）のメモリアクセス保護技術と連携した LMM を実装し、性能評価を実施し

た。また、ダミーデータを付与した状態でライセンス管理モジュールの処理時間を測定した。測定環境は表７のとおりである。結果を図13に示す。その結果、ダミーを追加した場合でも、十分小さいオーバーヘッドでライセンス管理モジュールが動作することを確認し、安全な実行プログラム構築技術を確立することができた。

表5　ライセンス管理モジュール評価環境

	PC	クラウド
OS	CentOS 6.3	Amazon EC2
CPU	Intel Core i7-3720QM 2.6GHz	Xeon 2.66GHz
RAM	8GB	

表6　ライセンス管理モジュール評価結果

	処理時間[ms]	
機能	PC	クラウド
ライセンス読み込み	43	42
ライセンス検証	37	37
ローカル環境検証	36	36
ライセンス移譲（AKEは除く）	83	156

表７　ダミーデータありの測定環境

OS	Windows XP Professional Version 2002 SP3
CPU	Intel(R) Core2 Duo CPU E8600 @ 3.33GHz
メモリ	3.49 GB

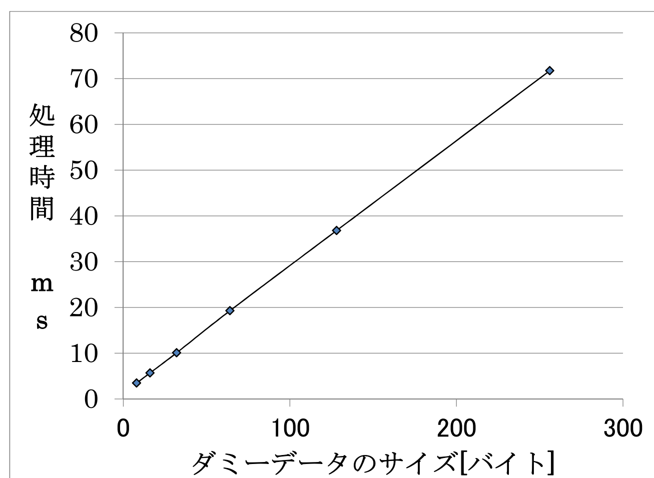


図13　ダミーデータありの場合の処理時間

３．今後の研究開発成果の展開及び波及効果創出への取り組み

本研究開発ではメモリアクセスパターン保護手法を確立し、その性能評価を行った。実用化に向けては、十分な安全性検証が不可欠であり、今後、実環境での安全性検証および理論的な安全性検証に取り組む予定である。また、アクセスパターンのみならず、データそのものの保護も重要である。アクセスパターン保護の性質から、従来の暗号アルゴリズムをそのまま適用することは出来ないため、アクセスパターン保護に特化したアルゴリズムの検討が必要である。アクセスパターン保護を暗号アルゴリズムと組み合わせて実現出来れば、ソフトウェアのみで高い安全性を有するプラットフォームを構築することが可能であり、PCやスマートフォン、組み込み系システムなど、幅広いシステムの安全性を向上することが可能であり、実用化に向けてこれらの課題に取り組む。また、クラウドへのアクセスをクラウド事業者から秘匿するといった応用も可能であるため、適用先の一つとして検討を進める。本研究開発で確立したアクセスパターン保護と同等の技術の研究は盛んに行われているが、ほとんどが学術的な研究であり、プロトタイプシステムなどの作成も殆ど行われていない。したがって、本技術の実用化によって、日本の国際競争力の強化も可能である。

安全な実行プログラムの構築技術で確立したプログラム保護によって、あるサーバから別のサーバへライセンス情報を安全に引き継ぐプロトコルを実現した。これによって、正規のライセンス情報があれば複数あるサーバから任意のものを選択し、そのサーバでプログラムを実行させることが可能となった。本技術を拡張し、障害等によって普段利用しているサーバが利用できなくなった場合でも、別のサーバにライセンス情報を移行し、プログラムを引き続き実行可能とする技術の実現に向けた取り組みを進める。また、個別のサーバだけでなく、ネットワーク環境も含めたクラウド全体の信頼性や処理能力を評価し、複数あるクラウド環境からからユーザにとって最適な環境を選択し、その環境でユーザがプログラムを実行するプラットフォーム構築に向けた研究開発課題を今後の国際連携に向けた研究課題として提案した。

課題ア、イともに、国際会議での成果発表や国際特許出願による知的財産権の確保に向けた取り組みを行う。また、ロンドン大学・フラウンホーファーとの連携を続け、実用化に向けた検討を共同で行なっていく予定である。とくに、フラウンホーファーが参加しているSecFuturとの連携や、本研究開発の成果を活用したHorizon2020への提案など、積極的に活動を継続する。国際標準化団体における関連分野の動向調査を行い、本研究開発で得られた成果の提案にむけて準備を進める。

４．むすび

本研究開発では、ロンドン大学、フラウンホーファーSIT研究所と連携し、課題ア）安全なプログラム実行を実現するメモリ保護技術と課題イ）安全な実行プログラムの構築技術の二つに取り組んだ。

ア）安全なプログラム実行を実現するメモリ保護技術

近年、暗号実装の脆弱性を利用したサイドチャネル攻撃が大きな問題となっている。サイドチャネル攻撃はハードウェア実装をターゲットとしたものが主であるが、ソフトウェアに対するサイドチャネル攻撃も存在しており、メモリアクセスを解析し秘密情報を復元する攻撃が主要な攻撃である。メモリアクセスを攻撃者から保護する汎用的な手法として一般的であるのが、Oblivious RAM(ORAM)である。しかし、既存のORAMはオーバーヘッドが大きく、実用化は困難である。オーバーヘッドが大きくなる要因として、データの並び替えを行うプロセスがある。データの並び替えを含め、ORAMに関する多くの改良が提案されているが、実用化に向けてはさらなる改良が求められている。そこで、本研究開発では、実用的なメモリアクセス保護手法として、従来のORAMの課題であったデータの並び替えを不要とし、アクセス履歴を用いたアクセスパターンの保護手法を確立する。プログラムの実行中に一度しかアクセスされないデータに関しては、そのアクセスがプログラムによるアクセスであるのか、ランダムに生じたアク

セスであるのか、が判別できないため、アクセスパターンの漏えいを気にする必要がない。しかし、複数回アクセスされるデータに関しては、プログラムが必要としているデータである可能性が高く、なんらかの保護手法が必要となる。そこで、過去にアクセスしたデータの履歴を保存しておき、プログラムがメモリにアクセスする際に、必要としているデータに加えて履歴内のデータにもアクセスを行い、意図的に複数回アクセスすることで、本当に必要なアクセスを秘匿する。課題ア）では次の3点を目標として挙げ、研究開発にとりくんだ。

- アクセス履歴を用いたアクセスパターンの保護手法を確立する。
- 上記の保護手法において、アクセスに必要なオーバーヘッドを 5 倍以下に抑える高速化手法を確立する。
- 上記保護手法を実装し、有用性を検証する。

その結果として、アクセスパターン保護手法を確立し、理論的及びシミュレーションによってそのパフォーマンスを評価した。理論検証では、アクセスパターン保護に必要な演算の回数を評価し、それらの合計が 5 となることを示した。また、Linux 環境で本手法を実装し、アクセスパターン保護に要する時間を計測した。従来手法に比べて十分高速であり、1MB のデータに対するオーバーヘッドが 130ms であることを示した。

イ）安全な実行プログラムの構築技術

プログラムの解析を防止するための技術としては、難読化がある。難読化により、プログラムの解析コストをその価値以上に高くできれば、実質的に解析を防止できると考えられる。一方で、難読化できない関数の存在が示されており、その限界が明らかになりつつある。また、不正実行を防止するための技術としては、プログラムの実行時に、信頼できるデバイスにより実行プロセスを検証する方式がある。しかし、実行に必要なデータがすべて信頼できない環境上にある場合には、検証処理をバイパスすることで、プログラムを実行される可能性が残る。このリスクを軽減するためには、アプリケーションと信頼できる環境が連携し、プログラムを実行する必要がある。そこで、信頼できない環境上でプログラムの動作を保護する手法を検討し、有用性を評価する。

課題イ）では次の 2 点を目標として挙げ、研究開発に取り組んだ。

- 課題ア）の技術を組み込んだ信頼できない環境におけるプログラム保護手法を確立する。
- 上記保護手法を用いた仮想サービスを構築し、技術の有用性を検証する。

課題イ）では、起動毎にプログラムのライセンス情報を確認するシステムを構築した。これによって、不正なコピーの実行などを防止することが可能となり、信頼出来ない環境においてプログラムを保護することが可能となる。また、ライセンス情報を扱う処理では課題ア）の技術を利用することで、ライセンス情報が不正に抜き出されることがないようにした。本技術を用いて、簡単な数値計算を行うプログラムを実装し、正しく動作することを確認した。また、ライセンス情報の確認にかかるオーバーヘッドも数十 ms に抑えられており、システムが十分有用であることを確認した。

【誌上発表リスト】

[1] Shinsaku Kiyomoto, Andre Rein, Yuto Nakano, Carsten Rudolph, and Yutaka Miyake, “LMM: A Common Component for Software License Management on Cloud”, The 10th International Conference on Security and Cryptography, pp284-295, (July, 2013)

【申請特許リスト】

[1]]仲野有登、清本晋作、三宅優、記憶装置、アクセスパターンの秘匿方法およびプログラム、日本、2013 年 3 月 1 日

[2] 清本晋作、仲野有登、三宅優、ライセンス管理システム、方法及びモジュール、日本、2013 年 3 月 22 日