Vol.2014-OS-130 No.12
2014/7/28

仮想マシンモニタによるプログラムコードの秘匿化

平井 成海[1,a)]　髙橋 一志[1,b)]　大山 恵弘[1,c)]

概要：商用アプリケーションには多くの技術が含まれており，それらの技術の中には利用者から隠したいものが含まれている場合がある．アプリケーションのコードに含まれている技術を隠すための手段として，難読化と呼ばれる手法が広く使われている．難読化はコードを複雑にすることでコードの解析を妨害する手法であるが，ほとんどの難読化手法においては静的解析によってコードの挙動を解析されてしまうため，今日においては難読化以上の対策が求められている．そこで本研究では，仮想マシンモニタを利用することでアプリケーション内の関数を秘匿しながら実行できるシステムを提案する．提案システムでは，プログラマがコード内の秘匿化対象関数を指定して切り出すことで秘匿化を行う．秘匿化対象関数の本体は仮想マシンモニタ内に格納し，その他の部分は仮想マシン内に格納する．秘匿化対象関数を呼び出すための命令が実行されたときに，仮想マシンモニタで検知して秘匿化対象関数を仮想マシンモニタ内で実行し，仮想マシンに結果を返すことで秘匿化を実現する．仮想マシンと仮想マシンモニタは完全に隔離されているため，仮想マシンモニタ内部にある秘匿化対象関数の本体を仮想マシンから入手することは不可能である．提案システムは関数呼び出しのたびに仮想マシンと仮想マシンモニタ間の遷移が発生するため，オーバーヘッドの増大が懸念される．そのためどれだけのオーバーヘッドが生じるかの調査を行った．

Hiding Program Code Using Virtual Machine Monitor

Abstract: Commercial softwares contain a large amount of technologies. Software vendor may want to hide these techniques from malicious users so that protect software patents. There are many obfuscation techniques to interfere with code analyzing from malicious user. However, the obfuscation techniques are weak because of malicious users can even analyze the obfuscated source code using the static analyzing technique. Nowadays, we have to consider more complicated countermeasure techniques against malicious users. In this paper, we proposed a system allows us to run program codes with hiding these codes from end-users using Virtual Machine Monitor (VMM). In our system, if programmers instruct our complier program functions to hide, the complier extracts these functions and then insert into VMM. The rest of codes insert into Virtual Machine (VM) When the hiding function is called from VM, VMM detects the calling and then execute the hiding function as a result VMM returns the result of the code into the VM. Although malicious users try to obtain the hidden function code, they cannot get the code because of VM and VMM is completely isolated In our system, the hiding function calling generates context switches between VM and VMM as a result the system may suffer from the overhead.

1. はじめに

商用アプリケーションのコード内には多くの技術が含まれており，それらの技術の中には利用者から隠したいものが含まれている場合がある．アプリケーションのコードに含まれている技術を隠すための手段として，難読化 [1] と呼ばれる手法が広く使われている．難読化はコードを複雑にすることでコードの解析を妨害する手法であり，例えば制御フローを変えることで難読化を行う手法 [2] や変数や関数の識別子を複雑にすることで難読化を行う手法 [3] がある．しかし，制御フローやデータフローの解析により，難読化されたプログラムから情報を抽出する手法 [4] なども存在するため，難読化以上の対策が求められるケースもある．なぜならば，ほとんどの難読化手法においてはコードそのものを隠すことはできず，静的解析によって挙動を解析することは不可能ではない．そこで，秘密にしたい技術が含まれる部分のコードそのものを隠し利用者には入出力のみを見せることで，静的解析を困難にすることができ

[1] 電気通信大学
The University of Electro-Communications
[a)] n.hirai@ol.inf.uec.ac.jp
[b)] kazushi@inf.uec.ac.jp
[c)] oyama@inf.uec.ac.jp

© 2014 Information Processing Society of Japan

ると考えた．

本研究では，仮想マシンモニタを使用することでアプリケーション内の指定した関数コードを利用者から秘匿しながら実行できるシステムを提案する．提案システムでは，プログラマがコード内の秘匿化対象関数を指定して切り出すことで秘匿化を行う．切り出しには，コンパイラ基盤である LLVM に拡張を加えたプログラムがアプリケーションのソースコードを処理することにより実現する．秘匿化対象関数の本体は仮想マシンモニタ内に格納し，公開部分は仮想マシン内に格納する．秘匿化対象関数を呼び出すための命令が実行されたときに，仮想マシンモニタで検知して秘匿化対象関数を仮想マシンモニタ内で実行し，仮想マシンに結果を返す．仮想マシンと仮想マシンモニタの空間は完全に隔離されているため，秘匿化が実現できる．本研究では仮想マシンモニタとして KVM を使用する．現状では，提案システムの適用対象は 64bit x86 上で動作する Linux アプリケーションであるが，他のプラットフォームに適用できるように提案システムを拡張することは難しくないと考えている．実験として提案システムを使用して CRC32 チェックサムの計算と mruby によるフィボナッチ数列の計算を行い，計算時間のオーバーヘッドが最大約 170 倍になることを確認した．オーバーヘッドは大きいものの，今後このオーバーヘッドは削減できる可能性があるため，この点についてもあわせて論じる．

本論文の構成は以下の通りである．2 章で本研究の目的とユースケースについて述べる．3 章で提案システムの設計について述べる．4 章で提案システムの実装について述べる．5 章で提案システムを使用した実験について述べる．6 章で関連研究について述べ，7 章で本論文のまとめと今後の課題について述べる．

2. 目的とユースケース

レンダラーソフトウェアのような高価なソフトウェアには，利用者から隠したい技術が含まれていることが多い．本研究ではそれらのソフトウェアに含まれている隠したい技術を仮想マシンモニタを使用して利用者から秘匿し，知的財産権を守ることを目的とする．

提案システムの適用対象として，クラウド環境の利用形態の一つである IaaS (Infrastructure as a Service) がある．IaaS では，サービスを稼働させるための仮想マシンなどをネットワーク経由で提供する．将来的には IaaS クラウド環境下でユーザがアプリケーションを利用するような利用形態が普及すると考えられる．また，仮想マシンを提供するための仮想マシンモニタなどの管理は通常はクラウド事業者が行う．本研究では，アプリケーションベンダとクラウド事業者は信用できるとみなし，信用できないのは利用者のみであると仮定する．

本研究のユースケースを図 **1** に示す．プログラマは C 言語でアプリケーションの開発を行う．プログラマは秘匿化対象関数を列挙し，アプリケーションから切り出す．関数単位であるためプログラマが秘匿化したい部分を指定することが容易になる．その他のコードは公開部分として公開される．秘匿化対象関数を呼び出すコードは，仮想マシン–仮想マシンモニタ間の関数呼び出しを可能にするハイパーコールに置き換えられる．

提案システムを使用してソフトウェアを利用者に提供するためには，以下の手順を行う．

(1) アプリケーションの提供者はクラウド事業者と契約を結び，秘匿化対象コードと公開部分のコードを提供する
(2) クラウド事業者は仮想マシンモニタ内に秘匿化対象コード，仮想マシン内に公開部分のコードを配置する
(3) クラウド事業者は利用者に対して仮想マシンを提供する

利用者は提案システムを意識せずにアプリケーションを利用できる．仮に，悪意のある利用者がデバッガ等を利用してコードを解析しようとしても，手に入れられる逆アセンブル結果は公開部分のコードのみである．そのコード内では，秘匿化対象関数の呼び出しはハイパーコールに置き換えられている．利用者は仮想マシンモニタ以下にアクセスできないので，秘匿化対象コードの入手もできない．結果，利用者は関数の入力を変化させたときの出力の変化から内部コードを推測するよりほかなく，内部コードの逆アセンブル結果がわかっているときに比べ，コード解析がはるかに困難である．結果として，悪意のある利用者からコードを守ることが可能になる．

3. 設計

3.1 実行システム

仮想マシンモニタに改造を加えることで，秘匿化対象関数を仮想マシンモニタ内で実行可能にする．提案システムの実行システムの図を図 **2** に示す．仮想マシンから仮想マシンモニタに対してハイパーコールが実行されると，仮想マシンモニタに制御が移る．仮想マシンモニタは制御が移った理由を調べハイパーコールによるものであった場合，提案システムの呼び出しかどうかをチェックする．チェックが必要な理由は改造前の仮想マシンモニタでもハイパーコールは使用されており，それらと区別をするためである．

また，仮想マシンと仮想マシンモニタのメモリ空間は独立しており，通常の呼び出し規約に準拠した呼び出しは使用できない．そのため，秘匿化対象関数の情報を別途渡す必要がある．提案システムでは秘匿化対象関数を呼び出す際は仮想マシン側のコードが関数 ID を渡し，仮想マシンモニタ側のコードは渡された関数 ID から秘匿化対象関数

© 2014 Information Processing Society of Japan

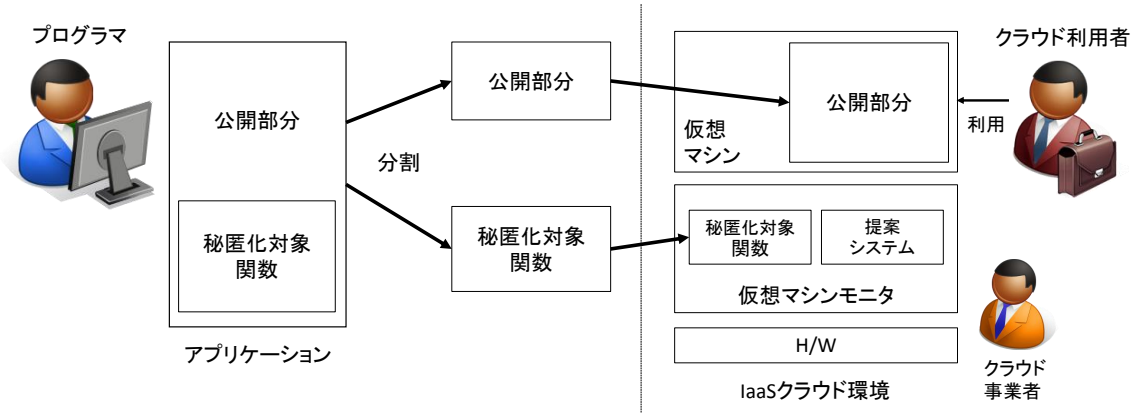

図 1　提案システムのユースケース
Fig. 1　Use case of the proposed system

の本体とその型情報を取得する．型情報は秘匿化対象関数の戻り値の型に関する情報である．それらを元にして仮想マシンモニタ内で秘匿化対象関数を実行し，戻り値を返却する．

3.2　秘匿化対象関数の本体

秘匿化対象関数の本体群はアプリケーションから切り出される．また，切り出した関数群を元にコンパイラとリンカによって共有ライブラリを作成し，`dlopen` 関数や `dlsym` 関数を使用して仮想マシンモニタのユーザレベル部分内に動的ロードすることで容易に取り扱うことができる．すなわち，秘匿化対象関数はホスト OS 上で動作する仮想マシンモニタのメモリ空間内にロードされ，実行する．そこで提案システムは，秘匿化対象関数の本体群と型情報を取得するための関数を共有ライブラリ内に自動的に含める．上述した通り，仮想マシンと仮想マシンモニタのメモリ空間は独立しており，通常の呼び出し規約に準拠した呼び出しは使用できない．そのため，関数が要求する呼び出し規約に準拠するための作業が必要である．また，その作業には型情報が必要であるため，型情報を取得するための関数を共有ライブラリに追加する必要がある．秘匿化対象関数内ではポインタによる仮想マシン側のメモリアクセスが可能である．また，秘匿化対象関数に関しては以下の仕様を定めた．

- 仮想マシン上のグローバル変数は直接使用できない（グローバル変数へのポインタを秘匿化対象関数の引数を通して渡した場合は可）
- 関数やシステムコールの呼び出しはできない
- `malloc` 関数などによるメモリの確保はできない（自動変数によるメモリの確保は可）
- アクセス違反が発生した場合，仮想マシン側には `NULL` ポインタに対するページフォルトが注入されること以外の動作は未定義である
- 秘匿化対象関数が利用できるスタックのサイズは，そのときの仮想マシンモニタの状態に依存する

秘匿化対象関数の本体は `__subst_` + 関数 ID (0 詰め) というシンボルでエクスポートされ，例えば関数 ID が `0x0123` であれば共有ライブラリにおけるシンボルは `__subst_0123` となる．

3.3　呼び出し規約

上述したように，仮想マシンと仮想マシンモニタのメモリ空間は独立しているため，秘匿化対象関数の情報を別途渡す必要がある．また，切り出した関数は 64bit Linux で使用されている System V AMD64 ABI に準拠しており，スタックの処理などが含まれているためそのままでは使用できない．そこで，秘匿化対象関数の呼び出し規約は実装を容易にするため System V AMD64 ABI に以下の変更の加えたものとした．

- スタック渡しの廃止（関数の引数はレジスタ渡しが可能な範囲に制限される）
- `EAX` レジスタの上位 16bit を `0xCA11`，下位 16bit を関数 ID にする（`0xCA11` は提案システムのハイパーコールを意味するマジックナンバーである）
- `call` 命令の代わりに `vmcall` 命令を使用する

3.4　秘匿化対象関数の切り出し

秘匿化対象関数の切り出しは，コンパイラに機能を追加することで実現する．コンパイラへの入力として，ソースコードの他に秘匿化対象関数名を記述した関数リストファイルを用意する．コンパイラは関数リストファイルを元に切り出す関数を決定し，切り出した関数を別のファイルに保存する．このとき，関数 ID を付与することで識別を可能にする．また，秘匿化対象関数を仮想マシンモニタを通して呼び出すようにするため，元のプログラムにおける秘匿化対象関数に対する呼び出しは関数 ID のロード命令とハイパーコールに置き換えられる．

© 2014 Information Processing Society of Japan

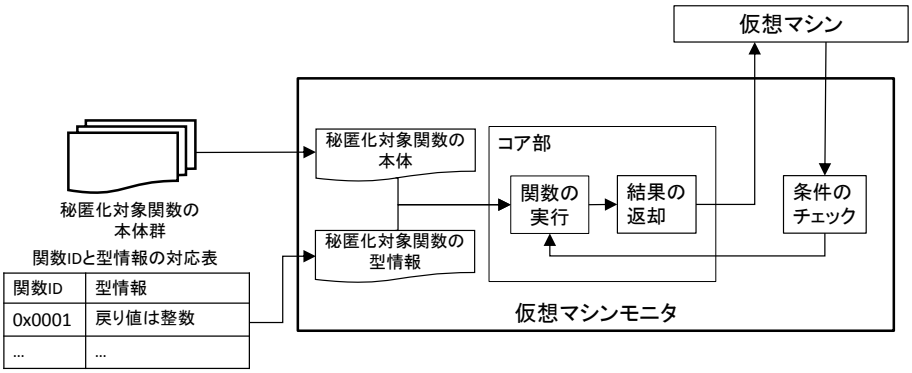

図 2　提案システムの実行システム
Fig. 2　Execution system of the proposed system

3.5　メモリアクセスのフック

仕様上は，秘匿化対象関数内ではポインタによるメモリアクセスを行うことはできる．しかし，アプリケーションが使用している仮想マシン側のポインタを仮想マシンモニタ側で直接メモリアクセスに使用することはできない．また，仮にポインタによるメモリアクセスが可能になったとしても，ページフォルトが発生した際に仮想マシンに通知しなければならないなど，複雑な処理が必要となる．本研究では，秘匿化対象関数内からの仮想マシン内のメモリへの全てのアクセスをフックすることによって問題を解決する．また，ページフォルトが発生した際には NULL ポインタにおけるページフォルトを仮想マシンに注入する．メモリアクセスのフックも，コンパイラへの機能追加によるプログラムの書き換えによって実現する．秘匿化対象関数内にメモリアクセスが発生する命令が含まれているか調べ，含まれていた場合はアドレス変換を行うためにフック関数を呼び出すようにその命令を書き換える．アドレス変換は仮想マシン内のプログラムにおける仮想アドレスから仮想マシンの仮想 RAM のアドレスへと仮想マシン内のページテーブルを参照して変換することで行う．

4.　実装

4.1　実行システム

秘匿化対象関数を仮想マシンモニタ内で実行するシステムを，カーネル空間で動作するドライバである KVM [5] とユーザ空間で動作する QEMU [6] に改造を加えて実装した．

4.1.1　KVM へのエミュレーション条件の追加

秘匿化対象関数を呼び出すためのハイパーコールは以下のように実装した．KVM の x86/vmx.c の kvm_vmx_exit_handlers 配列に Intel VT-x の VM Exit 理由別のハンドラ関数が登録されている．vmcall 命令が実行されたことによる VM Exit は EXIT_REASON_VMCALL で表され，ハンドラ関数は KVM の x86/vmx.c の handle_vmcall 関数である．vmcall 命令が秘匿化対象関数の呼び出しのために実行されたのであれば，仮想マシンの EAX レジスタの値は関数 ID が含まれた呼び出し用の値になっているはずである．よって，handle_vmcall 関数に以下の処理を追加した．

(1) 仮想マシンの EAX レジスタの値を読み込む
(2) 値の上位 16bit が 0xCA11 であるかを確かめる
(3) 上位 16bit が 0xCA11 であれば，QEMU へエミュレーションを要求する（そうでなければ，本来の仮想マシンモニタの処理へ移行する）

KVM に標準で用意されている kvm_register_read 関数を仮想マシンのレジスタの値を読み込むために使用した．QEMU へエミュレーションを要求する際には，vcpu->run->exit_reason にエミュレーションが必要な理由を代入し，handle_vmcall 関数の戻り値を 0 にする必要がある．秘匿化対象関数の実行要求を表すために，エミュレーション理由の値として KVM_EXIT_CALLFUNCTION という定数を用意し，それを使用した．

4.1.2　KVM への ioctl リクエストの変更と追加

KVM への ioctl リクエストは QEMU から KVM に対して仮想マシンの実行や汎用レジスタの読み書きを行うために利用される．これは提案システムを実装する上では不十分であったので，仕様の変更と新たな ioctl リクエストの追加を行った．

KVM の KVM_TRANSLATE ioctl リクエストは指定された仮想アドレスを物理アドレスへ変換するリクエストであるが，仮想アドレスの属するページが物理メモリ内にロードされているかのみチェックしているので，以下の条件をすべて満たしていない場合は変換を失敗させる（物理アドレスとして UNMAPPED_GVA を返す）ように変更した．

- ページが物理メモリ内にロードされていること
- ユーザ空間のページであること
- 読み書きが可能なページであること

KVM には kvm_inject_page_fault 関数という仮想マシンに対してページフォルトを発生させる関数が存在するが，ioctl リクエストとして外部から呼び出すことができないため，このままでは秘匿化対象関数が実行され

© 2014 Information Processing Society of Japan

る QEMU から呼び出すことができない．そこで，指定された仮想アドレスにおいてページフォルトを発生させる KVM_INJECT_PAGE_FAULT ioctl リクエストを追加することで QEMU から呼び出すことができるようにした．

4.1.3 QEMU へのコア部の実装

秘匿化対象関数の実行要求が KVM から QEMU にエミュレーション要求として伝えられると，コア部へと処理が移る．コア部の初期化が行われていない場合は初期化を行う．これは QEMU の仮想 RAM の先頭アドレスを取得するためのものである．次に QEMU に標準で用意されている kvm_getput_regs 関数と kvm_get_fpu 関数を呼び出して汎用レジスタを QEMU 側へ読み込み，EAX レジスタの値から関数 ID を取得し，関数 ID から対応する型情報を取得する．また，関数 ID に対応する秘匿化対象関数本体へのポインタを dlopen 関数と dlsym 関数で取得する．秘匿化対象関数の実行の前に，setjmp 関数を実行して秘匿化対象関数内でページフォルトが発生した際に制御を戻すための準備を行う．仮想マシンの汎用レジスタから仮想マシンモニタの汎用レジスタへ引数の値を読み込み，秘匿化対象関数を実行する．実行が終了したら，戻り値を仮想マシンの汎用レジスタへ書き込むために kvm_getput_regs 関数と kvm_set_fpu 関数を呼び出し，処理を終了する．

4.2 コンパイラへの機能の追加

提案システムに対応したコンパイラを，LLVM [7] にプラグインによって機能を追加することで作成した．LLVM はコンパイラを作成するために必要な要素を提供するコンパイラ基盤である．プラグインは LLVM を拡張するための仕組みであり，C++で開発され共有ライブラリの形としてビルドされる．また，C 言語のコンパイラフロントエンドとして Clang [8] を使用する．C 言語コードは LLVM+Clang でコンパイルすることで中間コード（IR）になり，後からプラグインによって見たり書き換えたりすることができる．また，IR からバイナリコードを生成することができる．

4.2.1 秘匿化対象関数の切り出し

秘匿化対象関数の切り出し方法として，河合らの作成したオブジェクトファイルを細分化するための LLVM プラグイン [9] を改造して使用した．このプラグインは関数を 1 つ 1 つ分割して動的ロードできる形にすることで，プログラムの読み込みにかかる時間を短縮するなどといったことを可能にするためのものである．本研究ではこのプラグインを改造することで秘匿化対象関数を指定しての切り出しと切り出した秘匿化対象関数を実行できるようにするためのハイパーコールへの置き換えを可能にした．

LLVM において，IR に対する処理は Pass という形で実装される．Pass にはいくつかの種類があり，モジュール内の関数に対する Pass（FunctionPass）や関数内の基本ブロックに対する Pass（BasicBlockPass）などがある．これらを継承して処理を実装することで独自の Pass を実装することができる．また，複数の Pass をシェルにおけるパイプによる文字列処理のように組み合わせ，複雑な処理を行わせることもできる．上記の LLVM プラグインにおける処理や，後述するメモリアクセスのフックのために IR を書き換えるための処理は Pass の 1 つとして実装されている．

4.2.2 メモリアクセスのフック

LLVM IR において，メモリアクセスが発生する命令は組み込み処理を除くと load 命令と store 命令の 2 つである．この 2 つの命令に対してフックコードを挿入すれば，メモリアクセス前にアドレスを変換し，ポインタを利用する上での問題を解決することが可能になる．本研究では，このフックコードを挿入するための Pass を作成した．フックコードではアドレスの変換関数が呼び出される．アドレスの変換関数の宣言を以下に示す．

```
void *__trans_memory__(void *linear);
```

この関数は MMU と同様の処理を行う．すなわち，仮想マシン上のハイパーコールの呼び出し元のプログラムにおける仮想アドレスを受け取り，仮想マシンのページテーブルを参照して仮想マシンモニタ内で使用できる仮想 RAM における該当ページのアドレスを返す．もし，ページアウトしているページであったり読み書きできないページであったりした場合など，不適切な状態であった場合にはプログラムを終了させるために NULL ポインタに対するページフォルトを注入する必要があるため，longjmp 関数によって例外を発生させる．

LLVM IR では void 型ポインタは 8bit 整数のポインタ（i8*）として表される．データ型を合わせるために，フック関数にポインタを渡すためには元のポインタ型から void 型ポインタへキャストし，変換後に元のポインタ型へ戻す必要がある．LLVM IR において，ポインタ型のキャストは bitcast 命令で行う．

また，C 言語の自動変数のためのメモリ領域は，LLVM IR では alloca 命令で確保される．秘匿化対象関数の自動変数のためのメモリ領域も同様に alloca 命令で確保されるが，このメモリに対してのアクセス，ポインタ型のキャスト（bitcast 命令），添字からのアドレスの計算（getelementptr 命令）についてはフック関数によるメモリアドレスの変換は不要である．なぜならば，自動変数は関数内に実行が移ることで初めて確保される変数であり，提案システムでは秘匿化対象関数の自動変数は仮想マシンモニタ内に確保されるからである．本研究で作成したフック関数を挿入する Pass を図 **3** の LLVM IR に適用すると，図 **4** の LLVM IR が得られる．図 4 における!exclude !0 は，その命令で行われる処理が alloca 命令で確保された

© 2014 Information Processing Society of Japan

```
%1 = alloca i32*, align 8
store i32* %p, i32** %1, align 8
%2 = load i32** %1, align 8
store i32 42, i32* %2, align 4
ret void
```

図 3　フック挿入前の LLVM IR
Fig. 3　LLVM IR before inserting the hook

```
%1 = alloca i32*, align 8
store i32* %p, i32** %1, align 8, !exclude !0
%2 = load i32** %1, align 8, !exclude !0
%varPtr = bitcast i32* %2 to i8*
%resultPtr = call i8* @__trans_memory__(i8* %varPtr)
%castPtr = bitcast i8* %resultPtr to i32*
store i32 42, i32* %castPtr, align 4
ret void
```

図 4　フック挿入後の LLVM IR
Fig. 4　LLVM IR after inserting the hook

表 1　実験環境
Table 1　Experimental environment

CPU	Intel Core i7 2600K
ホスト OS	Gentoo Linux 64bit
ゲスト OS	Gentoo Linux 64bit
ホストメモリ	16GB
ゲストメモリ	2GB
ホストカーネルのバージョン	3.12.21-gentoo-r1
ゲストカーネルのバージョン	3.12.21-gentoo-r1
KVM のバージョン	3.16
QEMU のバージョン	2.0.50

メモリに対するメモリアクセスであることを示すメタデータである．

4.2.3　提案システムにおけるコンパイルの流れ

提案システムによるアプリケーションのビルドは，以下の流れで行われる．

(1) プログラマは LLVM+Clang によってソースコードを LLVM IR へと変換する
(2) LLVM IR 対して秘匿化対象関数を切り出すプラグインを適用する（秘匿化対象関数のリストを与える）
(3) (2) で切り出された LLVM IR に対して，メモリアクセスをフックするための書き換えを行うプラグインを適用する
(4) (3) で得られた LLVM IR を実行可能なバイナリへと変換する

5.　実験

提案システムのオーバーヘッドを評価するために，2 つの実験を行った．実験は，表 **1** の環境で行った．

5.1　オーバーヘッドの測定

オーバーヘッドの測定は，CRC32 チェックサムの計算プログラムのチェックサムを計算する関数を以下の 3 つの状態において実行することによって行った．

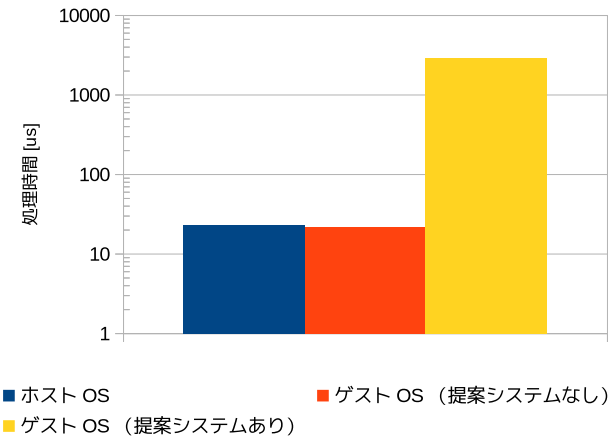

図 5　チェックサムを計算するプログラムの処理時間
Fig. 5　Processing time of a checksum calculation program

(1) ホスト OS 上で直接プログラムを実行したとき
(2) ゲスト OS 上で提案システムを使用しないでプログラムを実行したとき
(3) ゲスト OS 上で提案システムを使用してプログラムを実行したとき

4KB のデータに対する CRC32 チェックサムの計算を 10 回行い，それぞれの処理時間を平均したものをその状態の処理時間とした．処理時間の測定結果を図 **5** に示す．図 5 より，提案システムを使用した場合は使用しない場合に比べて約 130 倍多くの時間がかかったことがわかった．また，ホスト OS 上で実行したときよりもゲスト OS 上で提案システムを使用しないで実行したときの方が処理時間が短いが，これはホスト OS 上で動いている他のプログラムによるコンテキストスイッチににによるものである可能性がある．

オーバーヘッドを生じる要因を調べるために，提案システムを以下の 4 つに分けてそれぞれの処理時間を測定した．

(1) VM Exit や QEMU の処理などの仮想マシンモニタの処理
(2) 秘匿化対象関数の実行前後の処理
(3) 秘匿化対象関数の実行処理
(4) メモリアクセスによって発生したアドレスの変換処理

それぞれの処理時間の測定結果を図 **6** に示す．図 6 より，オーバーヘッドの大半はメモリアクセスによって発生したアドレスの変換処理によるものであることがわかった．実験に使用したプログラムでは 4KB のデータに対して CRC32 チェックサムを計算するためには 8192 回のアドレス変換が必要なため，1 回のアドレス変換にかかる時間は約 0.35[μs] である．アドレスの変換処理のオーバーヘッドが大きい原因は，提案システムはアドレスの変換結果のキャッシュなどをしておらず，毎回 KVM への `ioctl` の呼び出しとそれに伴うコンテキストスイッチが発生するためと考えられる．

© 2014 Information Processing Society of Japan

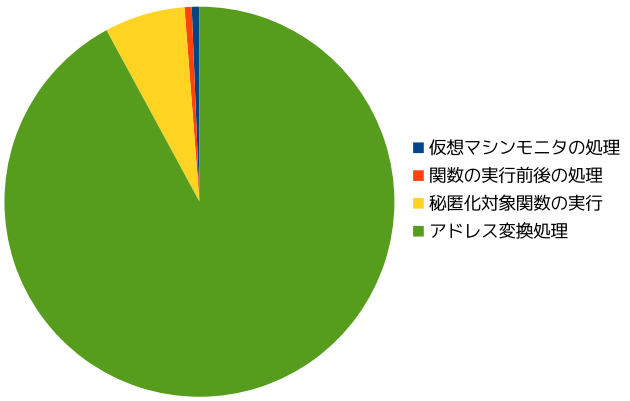

図 6　提案システムにおける処理時間の内訳
Fig. 6　Breakdown of processing time consumed by the proposed system

5.2　mruby による実験

提案システムを実用的なプログラムに適用する例として，mruby [10] に対して提案システムを適用する実験を行った．mruby とはプログラミング言語 Ruby の軽量な実装であり，Ruby 1.9 の文法と互換性がある．この実験では，フィボナッチ数列を計算するプログラムを Ruby で記述し，mruby 上で実行した．フィボナッチ数列の第 30 項の計算を 5.1 節に示した 3 つの状態において 10 回ずつ行い，それぞれの処理時間を平均したものをその状態の処理時間とした．また，mruby 内の 32bit 整数の加減算処理を行う関数を秘匿化対象関数として切り出した．それぞれの処理時間の測定結果を図 **7** に示す．図 7 より，提案システムを使用した場合は使用しない場合に比べて約 170 倍多くの時間がかかったことがわかった．5.1 節の実験よりもオーバーヘッドが増大したのは，32bit 整数に対する加減算の処理のたびに提案システムによる関数の実行が発生していることや，計算結果を格納するためにメモリアクセスが発生していることが原因と考えられる．フィボナッチ数列を計算するプログラムの実行では秘匿化対象関数が呼び出される頻度が極めて高かったためオーバーヘッドが大きくなったが，より現実的なプログラムでは秘匿化対象関数が呼び出される頻度はずっと低いため，オーバーヘッドはここまで大きくならないと予想される．

6.　関連研究

6.1　HyperCensor

佐久間らの HyperCensor [11] は，OS カーネルの命令を秘匿するために仮想マシンモニタを使用している．HyperCensor では，OS カーネル内の秘匿化対象命令を特権命令である `hlt` 命令で置き換え，`hlt` 命令の実行を仮想マシンモニタでトラップする．仮想マシンモニタ内では，秘匿化対象命令のシングルステップ実行を行い，結果を仮想マシンに戻すことで秘匿化を実現している．

本研究は仮想マシンモニタを使用するという点においては同一であるが，ユーザ空間で動作するプログラムに対して秘匿化を行える点と，秘匿化の細かさを関数単位にすることで秘匿化されたコードの効率的な実行を実現している点において異なる．

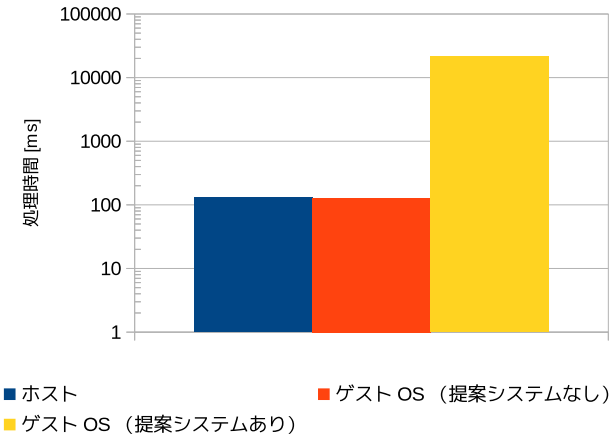

図 7　mruby 上でフィボナッチ数列を計算するプログラムの処理時間
Fig. 7　Processing time of a program that calculates a Fibonacci number on mruby

6.2　Web API

Web API はアプリケーションが Web サービスと連携を行うためのインターフェースであり，Web API を実現するための代表的な技術として SOAP や REST がある．Web API は入力を受け取りそれを元に出力を返す形を取っており，Web API は関数単位と考えることができる．また，具体的な内部挙動を公開しない限り静的解析が不可能である．よって，Web API も関数単位での秘匿化が可能であると考えることができる．Web API の呼び出しには関数呼び出しに比べて非常に長い時間がかかると考えられるため，繰り返し呼び出すようなユースケースは適さない．また，Web サービス側からアプリケーションのメモリを直接アクセスすることはできない．

同様の仕組みは，Web が普及する以前から RPC [12] として広く利用されていた．RPC は，遠隔の計算機が提供する手続きを呼び出すことを可能にする仕組みである．RPC を利用すれば，Web API と同じく，秘匿化したい処理をユーザから隠すことが可能である．しかし，RPC を使用しても Web API と同様の問題が生じる．

本研究は秘匿化の細かさが関数単位であるという点においては同一であるが，本研究は繰り返し呼び出すようなユースケースでも効率的な実行を行うことができる．また，ポインタを経由して秘匿化対象関数から公開部分のメモリにアクセスできるため，例えば行列データなどを形式の変換なしに秘匿化対象関数内で扱うことが可能である点においても異なる．オーバーヘッドにおいても，提案システムに優位性があると考えられる．

© 2014 Information Processing Society of Japan

6.3 Privtrans

Brumley らの Privtrans [13] は権限分割のためにプログラムを 2 つに分割し，お互いを RPC を使用して協調させて動作させている．分割は C 言語のソースコードを解析して変更を加えることで実現している．ユーザは，`__attribute((priv))__`で表される秘匿化を指示する属性を関数やグローバル変数に付加することで一般ユーザ権限で動作させるプログラムとスーパーユーザ権限で動作させるプログラムに分割することができる．一般ユーザからスーパーユーザのプログラムを見ることはできないので，一般ユーザによる静的解析は不可能である．よって，一般ユーザに対する関数単位の秘匿化が可能であると考えることができる．また，Privtrans では特権リソースに対するポリシーを設定できるが，本研究ではそのような機能は実装していない．しかし，本研究ではスーパーユーザで実行されるプログラムにおいてもコードを秘匿することができる．

6.4 SecureQEMU

William らの SecureQEMU [14] はアプリケーションコードの解析を防ぐために暗号化を使用しており，QEMU を拡張することで暗号化したコードをそのまま QEMU 上で実行することが可能である．また，悪意のあるユーザが暗号化されたコードを解析して内部の技術を知ることができる可能性があるので，SecureQEMU が行う処理は難読化の一種と考えることができる．

本研究は対象がアプリケーションである点においては同一であるが，本研究は静的解析を防止するために，ユーザが見ることができるのは秘匿化対象関数の呼び出し部のみであるという点において異なる．

7. まとめと今後の課題

本研究では，関数単位でアプリケーションのコードを秘匿化するシステムの提案と実装を行った．また，提案システムを CRC32 チェックサムを計算するプログラムや mruby に対して適用し，オーバーヘッドを測定した．測定の結果として，最大で約 170 倍のオーバーヘッドが発生することがわかった．ただし，このオーバーヘッドは秘匿化対象関数の実行頻度が今回の実験では非常に高かったことによるものであり，より現実的なプログラムではオーバーヘッドはより小さくなることが予想される．

今後の課題として，オーバーヘッドの削減やメモリアドレスの変換における読み/書きの区別による制限の緩和などが挙げられる．オーバーヘッドの削減については，オーバーヘッドの原因がコンテキストスイッチとアドレス変換であると考えられるので，コンテキストスイッチを発生させないためにアドレス変換を QEMU 内で完結させるという対策や，TLB のようなアドレス変換キャッシュを導入するという対策が考えられる．また読み/書きの区別については，現時点では変換対象の仮想アドレスはこれから読まれるのか書かれるのかを区別していないため，読み/書き共にできるページでないとアクセス違反によるページフォルトが発生する．秘匿化対象関数の本体は信頼できるので，これから実行される命令がメモリを読むのか書くのかについての情報をアドレス変換の際に利用するようにすれば，上述した制限はなくなると考えられる．

謝辞　本研究は JSPS 科研費 26330080 の助成を受けている．

参考文献

[1] Christian Collberg and Jasvir Nagra: *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*, Addison-Wesley Professional (2009).
[2] Chen, H., Yuan, L., Wu, X., Zang, B., Huang, B. and Yew, P.-c.: Control Flow Obfuscation with Information Flow Tracking, *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, New York, NY, USA, ACM, pp. 391–400 (2009).
[3] Tyma, P.: Method for renaming identifiers of a computer program, US Patent 6,102,966 (2000).
[4] Udupa, S. K., Debray, S. K. and Madou, M.: Deobfuscation: Reverse Engineering Obfuscated Code, *Proceedings of the 12th Working Conference on Reverse Engineering*, WCRE '05, Washington, DC, USA, IEEE Computer Society, pp. 45–54 (2005).
[5] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin and Anthony Liguori: kvm: the Linux Virtual Machine Monitor, *Proceedings of the Linux Symposium* (2007).
[6] QEMU: `http://www.qemu.org/`.
[7] Lattner, C.: LLVM: An Infrastructure for Multi-Stage Optimization, Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL (2002).
[8] Clang: `http://clang.llvm.org/`.
[9] 河合 夏輝，笹田耕一：オブジェクトファイルの細分化機構と応用，情報処理学会研究報告.，Vol. 2012-EMB-24, No. 21, pp. 1–6 (2012).
[10] mruby: `https://github.com/mruby/mruby`.
[11] 佐久間 充，大山 恵弘：HyperCensor: 仮想マシンモニタを用いた OS バイナリコードの秘匿化，ディペンダブルシステムワークショップ＆シンポジウム (DSW & DSS 2011), 京都工芸繊維大学 (2011).
[12] Birrell, A. D. and Nelson, B. J.: Implementing Remote Procedure Calls, *ACM Trans. Comput. Syst.*, Vol. 2, No. 1, pp. 39–59 (1984).
[13] Brumley, D. and Song, D.: Privtrans: Automatically Partitioning Programs for Privilege Separation, *Proceedings of the 13th USENIX Security Symposium*, Berkeley, CA, USA, USENIX Association, pp. 5–5 (2004).
[14] Kimball, W. B.: SecureQEMU: Emulation-based Software Protection Providing Encrypted Code Execution and Page Granularity Code Signing, Master's thesis, Air Force Institute of Technology (2008).

© 2014 Information Processing Society of Japan