Intel SGXとSMMを組み合わせた安全なホストベースIDSの実行機構

古賀　吉道[1]　光来　健一[1]

概要：システムの脆弱性を完全に取り除くことは困難であるため，侵入検知システム（IDS）を用いて対象システムを監視する必要がある．しかし，システムの異常を検知するホストベース IDS は監視対象システム上で動作するため，安全に実行することが難しい．例えば，システムが改ざんされると IDS が正確なシステム情報を取得することはできなくなる．これまで，汎用 CPU の機能を用いて安全に IDS を実行する様々な手法が提案されてきたが，安全性や柔軟性などの面で問題があった．本稿では，Intel SGX とシステムマネジメントモード（SMM）を組み合わせることで，より安全かつ柔軟に IDS を実行することが可能なシステム SSdetector を提案する．SSdetector は IDS を保護するために，SGX が提供する隔離実行環境のエンクレイヴ内で IDS を実行する．エンクレイヴ内ではシステムのメモリデータを安全に取得することができないため，SMM で動作する BIOS 内のプログラムを呼び出してメモリデータを取得する．SGX 仮想化をサポートした KVM を用いて VM 内に SSdetector を実装し，IDS が監視に用いる proc ファイルシステムに必要なシステム情報を取得する性能を調べた．

1. はじめに

近年，インターネットに接続された情報システムへの攻撃が数多く報告されている．攻撃の糸口となるシステムの脆弱性を完全に取り除くことは難しいため，侵入検知システム（IDS）を用いて対象システムを監視し，管理者に攻撃を通知する必要がある．しかし，システムの異常を検知するホストベース IDS は監視対象システム上で動作するため，安全に実行することが難しい．例えば，システムが改ざんされた後は，IDS がシステムから情報を取得したとしてもその情報が正しいという保証はない．また，IDS が改ざんされると無力化されてしまい，以降の攻撃を検知できなくなる．

これまで，汎用 CPU の機能を用いて安全にホストベース IDS を実行する様々な手法が提案されてきたが，安全性や柔軟性などの面で問題があった．例えば，Intel や AMD の CPU の動作モードの 1 つであるシステムマネジメントモード（SMM）を用いた IDS が提案されている [1]．この IDS は SMM のための独立した実行環境で安全に動作するが，IDS の管理が容易ではなく，性能面での影響も大きい．また，Intel SGX を用いた IDS も提案されている [2]．この IDS はエンクレイヴと呼ばれる隔離実行環境で安全に動作するが，仮想マシン（VM）のメモリデータを取得するために脆弱性が数多く報告されているハイパーバイザに頼る必要がある．

これらの問題を解決するために，本稿では SGX と SMM を組み合わせることでより安全かつ柔軟な IDS 実行を可能にするシステム SSdetector を提案する．SSdetector は SGX のエンクレイヴ内で IDS を実行し，IDS の盗聴や改ざんを防ぐ．IDS を OS 上で動作する SGX アプリケーションとして作成することにより，IDS の管理を容易にすることができる．一方，エンクレイヴ内ではシステムのメモリデータを安全に取得することができないため，SMM で動作する BIOS 内のプログラムを呼び出してメモリデータを取得する．IDS に共通の処理だけを SMM で実行することにより，IDS およびシステムの性能低下を抑えることができる．IDS と SMM プログラム間では，メモリデータの暗号化および整合性検査を行うことで盗聴や改ざんを防ぐ．

SGX 仮想化をサポートした KVM を用いて SSdetector を VM 内に実装した．SMM プログラムは UEFI BIOS のオープンソース実装である TianoCore に実装した．SMM プログラムはシステムマネジメント割り込み（SMI）を用いて呼び出され，OS データの仮想アドレスを物理アドレスに変換してメモリデータを取得する．SSdetector を用いて実験を行い，IDS が監視に用いる proc ファイルシステムに必要なシステム情報を取得できることが確認できた．

[1] 九州工業大学
Kyushu Institute of Technology

また，メモリ上のシステム情報を取得するのにかかる時間を測定し，SGX や SMM，暗号化，整合性検査を用いることによって生じるオーバヘッドを調べた．

以下，2 章でホストベース IDS の従来の安全な実行手法について述べる．3 章で Intel SGX と SMM を組み合わせることで IDS を安全に実行可能にするシステム SSdetector を提案する．4 章で SSdetector の実装について述べる．5 章で SSdetector の性能を調べた実験について述べる．6 章で関連研究に触れ，7 章で本稿をまとめる．

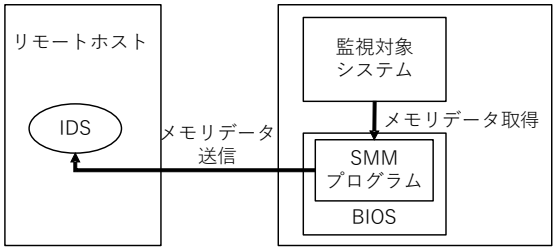

図 1 SMM を用いた IDS

2. ホストベース IDS の安全な実行

ホストベース IDS を安全に実行するための要件として，(1) 監視対象システムの機能を用いずに攻撃を検知できる，(2) 攻撃者に侵入されても IDS は改ざん・盗聴されない，(3) IDS が攻撃者によって停止されない，もしくは停止されたことを検知できる，の 3 点が挙げられる．監視対象システムの機能を用いてシステムの監視を行うと，システムが攻撃を受けた後は IDS がそのシステムから正しい情報を取得できる保証はなくなる．IDS が改ざんされるとそれ以降の攻撃を正しく検知できなくなる．IDS が取得したシステム情報を盗聴されると，システムへの攻撃に成功しなくても IDS から機密情報の一部を盗むことができる．また，IDS は攻撃を受けずに動き続けることが望ましいが，少なくとも IDS が停止させられたことを検知できる必要がある．

これまで，汎用 CPU の機能を用いてこれらの要件を満たす IDS が提案されてきたが，安全性や柔軟性などの面で問題があった．例えば，Intel や AMD の CPU によって提供されている SMM を用いて，監視対象システムのメモリから直接，システム情報を取得する IDS が提案されている [1], [3]．SMM は BIOS によってのみ使用可能な CPU の動作モードであり，OS であってもアクセスができない独立した環境を提供する．SMM で動作するプログラムは SMRAM と呼ばれる専用メモリに置かれ，SMM でしかアクセスすることができない．SMM で動作するプログラムは SMI と呼ばれるソフトウェア割り込みを発生させることにより呼び出される．SMI は特定の I/O ポートに書き込むことなどによって発生させることができる．

IDS を SMM プログラムとして実行することで，安全に実行するための要件を満たすことができる．HyperGuard [1] は SMM で動作する IDS がハイパーバイザのメモリを監視することにより改ざんを検知する．IDS が監視対象システムへの侵入者からの攻撃を受けることはない．しかし，IDS 全体を BIOS の一部として動作させるため，IDS の更新や新しい IDS の利用が容易ではない．また，SMM でのプログラム実行は低速であり，実行中はシステム全体が停止するため，IDS やシステムへの影響が大きい．そこで，HyperCheck [3] は図 1 のように SMM でネットワークドライバのみを動作させ，メモリデータをリモートホストに送信して監視を行う．この手法は IDS の管理をリモートホストで柔軟に行うことができ，性能への影響を最小限に抑えることができる．その一方で，リモートホストで動作する IDS は保護されていないため，IDS の安全性が課題となる．

また，Intel SGX を用いて仮想マシン（VM）のメモリから直接，システム情報を取得する IDS も提案されている [2]．SGX はアプリケーション内にエンクレイヴと呼ばれる隔離実行環境を作成することができる CPU のセキュリティ機構である．エンクレイヴ内のプログラムは CPU の保護により安全に実行することができる．プログラムの実行開始時には SGX によって電子署名が検査されるため，攻撃者によって改ざんされたプログラムは実行することができない．また，SGX によってエンクレイヴのメモリの整合性が保証されるため，攻撃者はエンクレイヴ内で実行中のプログラムを改ざんすることができない．加えて，エンクレイヴのメモリは暗号化されるため．エンクレイヴ内のデータが攻撃者によって盗聴されることもない．

IDS を SGX のエンクレイヴ内で動作させることにより，安全な実行の要件を満たすことができる．SGmonitor [2] では，図 2 のようにエンクレイヴ内の IDS が VM のメモリデータを取得して監視を行う．システムへの侵入者は IDS の改ざんや盗聴を行うことはできない．ただし，侵入者は IDS を動作させている SGX アプリケーションを容易に停止することができるため，リモートホストから IDS の正常動作を確認することで IDS の停止を検知する．エンクレイヴ内の IDS は VM の下で動作しているハイパーバイザ経由で VM のメモリデータを安全に取得する．そのため，ハイパーバイザに脆弱性があるとメモリデータを改ざんされたり盗聴されたりする恐れがある．また，ハイパーバイザを必要とするため，仮想化システムの監視しか行うことができない．

3. SSdetector

本稿では，SGX と SMM を組み合わせることにより，IDS をより安全かつ柔軟に実行できるようにするシステム SSdetector を提案する．

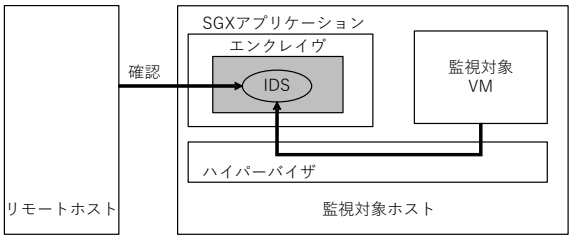

図 2 SGX を用いた IDS

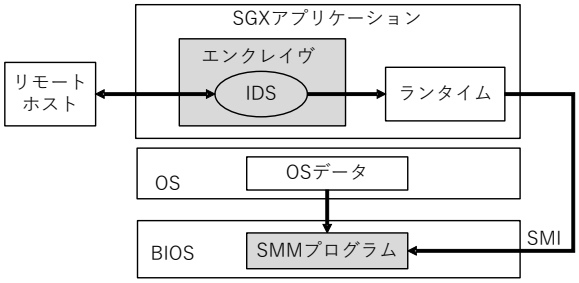

図 3 SSdetector のシステム構成

3.1 脅威モデル

SSdetector は CPU を信頼し，CPU に搭載されている SGX や SMM に脆弱性はないものする．また，BIOS 内で動作する SMM プログラムは信頼できるものとし，脆弱性はないものとする．ハイパーバイザと比べても SMM プログラムはサイズが非常に小さくインタフェースも狭いため，攻撃ははるかに難しい．BIOS はネットワーク経由での更新を許可しないように設定し，マシンに物理的にアクセスできる場合にだけ更新を許可する．そのため，監視対象システムに対する攻撃としては，外部の攻撃者によるネットワーク経由での攻撃を想定する．一方，BIOS 内の SMM プログラムと SGX エンクレイヴ内の IDS 以外のソフトウェアは信頼しない．

3.2 SGX と SMM を用いたメモリ監視

SSdetector は SGX と SMM を用いて監視対象システムのメモリデータを安全に取得することによりシステムの監視を行う．SGX が提供するエンクレイヴ内で IDS を実行し，CPU によるメモリの暗号化および整合性検査により IDS の盗聴や改ざんを防ぐ．IDS を更新する際には OS 上で動作する SGX アプリケーションを再起動するだけで済むため，柔軟に IDS の管理を行うことができる．一方，エンクレイヴ内ではシステムのメモリデータを安全に取得することができないため，IDS は SMM で動作する BIOS 内のプログラムを呼び出してメモリデータを取得する．SMM は独立した実行環境を提供するため，攻撃を受けることなく安全にシステムのメモリデータを取得できる．SMM では IDS 全体を動作させずメモリデータの取得のみを行うことで，SMM による性能低下を抑えることができる．

図 3 に SSdetector のシステム構成を示す．監視対象システム内で実行される SGX アプリケーションが IDS を動作させ，エンクレイヴと SSdetector ランタイムで構成される．ランタイムはエンクレイヴ内の IDS と SMM プログラムとの間の中継を行うプログラムである．ランタイムは SGX によって保護されておらず，攻撃を受ける可能性がある．BIOS 内にはシステムのメモリデータを取得する SMM プログラムが配置される．エンクレイヴ内の IDS の正常な動作を確認したり，検知結果を受け取ったりするためにリモートホストを用いる．

エンクレイヴ内の IDS がシステムを監視するために OS データを必要とした時には，OCALL と呼ばれる SGX の機能を用いてエンクレイヴの外部で動作する SSdetector ランタイムを安全に呼び出す．呼び出されたランタイムは SMI を発生させ，BIOS 内の SMM プログラムを呼び出す．ランタイム経由で SMM プログラムを呼び出すのは，エンクレイヴ内では SMI を発生させることができないためである．SMM プログラムを呼び出す際には OS データの仮想アドレスを渡す．ランタイム経由でアドレス情報が漏洩するのを防ぐために，SSdetector はエンクレイヴ内で仮想アドレスを暗号化する．

SMM プログラムは渡された仮想アドレスを物理アドレスに変換した後，メモリデータの取得を行う．これは SMM プログラムが物理アドレスでしかメモリにアクセスすることができないためである．IDS には OS データの仮想アドレスしか分からず，物理アドレスに変換しようとすると監視対象の OS の機能に依存してしまう．そこで，SMM プログラムはメモリ上の OS のページテーブルを探索することで安全にアドレス変換を行う．メモリデータが IDS に返されるまでの間に盗聴・改ざんされないように，SMM プログラムはメモリデータのハッシュ値を計算し，メモリデータを暗号化する．それらのデータは SSdetector ランタイム経由でエンクレイヴ内の IDS に返される．エンクレイヴ内ではメモリデータを復号し，そのハッシュ値を再計算して整合性検査を行う．

4. 実装

SSdetector の SMM プログラムを EDK II [4] を用いてオープンソースの UEFI BIOS である TianoCore に実装した．また，IDS を動作させる SGX アプリケーションを Intel SGX SDK 2.13.3 [5] を用いて実装した．実機の BIOS を変更するのは困難であるため，SSdetector を VM 内で動作させた．SGX をサポートした VM を作成できるようにするために，SGX 仮想化のためのパッチを適用した KVM SGX [6] と QEMU SGX [7] を用いた．

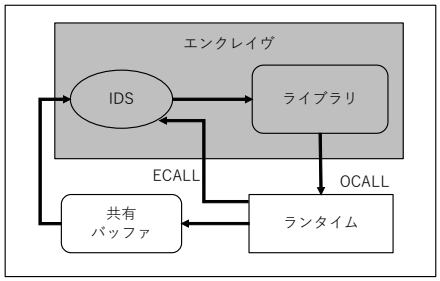

図 4 SGX アプリケーションの構成

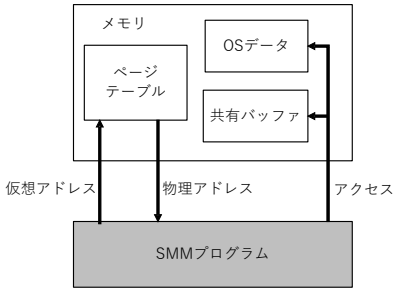

図 5 SMM プログラムにおけるアドレス変換

4.1 SSdetector ライブラリとランタイム

SSdetector の SGX アプリケーションの構成は図 4 のようになっている．IDS として用いられる SGX アプリケーションを起動すると，まず SSdetector ランタイムが実行され，IDS を動作させるためのエンクレイヴが作成される．IDS の実行を開始する際には，ECALL と呼ばれる SGX の機構を用いてエンクレイヴ内部の IDS の関数を安全に呼び出す．その際に，ランタイムとエンクレイヴの間でデータを受け渡すための共有バッファのアドレスを IDS に渡す．一方，エンクレイヴ内の IDS は SSdetector ライブラリ経由で OCALL を用いてランタイムを安全に呼び出す．

SSdetector ランタイムの OCALL 関数は，ioperm システムコールを用いて入出力許可を設定した上で 0xb2 番の I/O ポートに書き込みを行い，SMI を発生させる．SMM プログラムの中の SMI ハンドラに情報を渡すために，OS データの仮想アドレスと共有バッファの仮想アドレスをレジスタに格納する．その際に仮想アドレスを暗号化するが，それぞれのアドレスは 8 バイトしかないため，2 つの仮想アドレスを合わせた 16 バイトのデータに対して AES で暗号化を行う．そして，16 バイトの暗号データを 8 バイトずつに分割してそれぞれ RDI レジスタと RSI レジスタに格納する．

SMI ハンドラから戻ってくると，共有バッファに暗号化されたメモリデータとそのハッシュ値が格納されている．OCALL の呼び出しからエンクレイヴ内の SSdetector ライブラリに戻る際にも，この共有バッファを介してデータが渡される．ライブラリは共有バッファ上のメモリデータを復号してハッシュ値を再計算する．そして，共有バッファ上のハッシュ値との比較を行い，一致しなければ改ざんを検出する．復号したメモリデータはエンクレイヴ内でキャッシュすることにより，SMM プログラムの呼び出し回数を削減する．

メモリデータの暗号化に用いる AES の鍵はエンクレイヴ内の SSdetector ライブラリが生成する．生成した暗号鍵は SMM プログラムの RSA 公開鍵を用いて暗号化し，SMI ハンドラに渡す．その際に，0xb2 番の I/O ポートに書き込む値によってメモリデータの要求と区別する．SMM プログラムでは RSA 秘密鍵を用いて暗号鍵を復号する．暗号化されていない SGX アプリケーションのバイナリに格納されるのは SMM プログラムの公開鍵だけである．

4.2 SMM プログラム

SMM プログラムが監視対象システムのメモリ全体にアクセスできるようにするために，SSdetector では従来の BIOS ではなく，その後継である UEFI BIOS を用いる．UEFI BIOS は 64 ビットモードで動作し，4GB を超えるメモリにもアクセスできる．ただし，TianoCore においては SMM でアクセス可能なメモリ領域が制限されているため，SSdetector ではメモリ全体にアクセスできるようにこの制限を解除する．また，TianoCore は SMM のためのページテーブルを作成してメモリアクセスを制限しているため，SSdetector ではすべてのメモリにアクセスできるようにページテーブルを作成する．

SMI によって呼び出された SMM ハンドラはまず，CPU の RDI レジスタと RSI レジスタに格納された仮想アドレスを 16 バイトのデータとして復号し，2 つの仮想アドレスを取り出す．次に，CPU の CR3 レジスタの値を取得して OS メモリ上のページテーブルを特定する．そのページテーブルを用いて図 5 のように OS データの仮想アドレスを物理アドレスに変換し，共有バッファの仮想アドレスも物理アドレスに変換する．その後，OS のメモリデータを読み出し，そのハッシュ値を計算して共有バッファに格納する．また，メモリデータを暗号化して共有バッファに書き込む．

4.3 LLView の利用

SSdetector では LLView フレームワーク [8] を用いることで，Linux カーネルのソースコードを利用して OS データを監視する IDS を開発することができる．LLView は IDS をコンパイルして生成された LLVM の中間表現に対してプログラム変換を行い，IDS が透過的に監視対象システムのメモリにアクセスできるようにする．SSdetector では，IDS がシステムのメモリにアクセスする際に OCALL を呼び出すように LLView を修正した．

この LLView を用いて，Linux の proc ファイルシステムが提供するシステム情報を収集するホストベース IDS を開

```
/proc/stat:
cpu  71777 1194 21338 6577714 134423 0 128 134 0 0
cpu0 71777 1194 21338 6577714 134423 0 128 134 0 0
intr 1853674 0 8057 0 0 0 0 0 0 0 0 144 0 0 0 0 0 0 83 0 3176 39 0 1 1 1 1 1
6 0 63260 0 98901 0 36767 2839 33889 3389 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
ctxt 8321751
btime 0
processes 61953
procs_running 1
procs_blocked 0
softirq 3634260 65 1115298 21 39823 128307 0 6555 0 0 2344191
```

図 6 IDS が取得したシステム情報の出力例

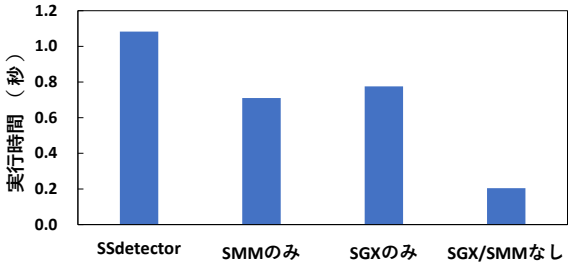

図 7 システム情報の取得時間

発した．proc ファイルシステムはプロセスやネットワークなどを含むシステムの情報を疑似ファイルとして提供しており，システムの監視によく用いられている．この IDS は Linux の proc ファイルシステムのソースコードを利用して，疑似ファイルの作成に必要な OS データを収集する．

5. 実験

SSdetector の有効性を調べるために，4.3 節で述べたホストベース IDS を用いて実験を行った．この IDS は Linux の proc ファイルシステムが提供するシステム情報のうち，9 種類の疑似ファイルを作成するのに必要なシステム情報を収集する．実験に使用したマシンの CPU は Intel Core i7-9700，メモリは 16GB であった．本実験では，SGX 仮想化をサポートした KVM SGX v5.6.0-rc5-r2 および QEMU SGX v4.0.0-r1 を用いて SSdetector を VM 内で動作させた．この VM には仮想 CPU を 1 個，メモリを 2GB 割り当て，VM 内では Linux 5.8.18 を動作させた．

5.1 IDS の動作確認

SSdetector を用いて IDS をエンクレイヴ内で実行し，SMM プログラム経由でメモリ上のシステム情報を収集して作成した疑似ファイルの内容を出力させた．図 6 に出力されたシステム情報の一部を示す．この出力結果を監視対象システムにおいて proc ファイルシステムの疑似ファイルの内容と比較したところ，システム情報が正しく取得できていることが確認できた．

5.2 SGX と SMM を用いるオーバヘッド

SSdetector を用いて IDS を実行し，必要なすべてのシステム情報を収集するのにかかる時間を測定した．SGX と SMM を用いる SSdetector との比較として，(1) SGX を使用せず SMM のみを使用する場合，(2) SMM を使用せず SGX のみを使用する場合，(3) SGX と SMM のどちらも用いない場合についても測定を行った．SGX を用いない場合は，OCALL を呼び出す代わりに直接，SSdetector ランタイムを呼び出すようにした．SMM を用いない場合は，SMI を発生させる代わりに OS に組み込んだデバイスドライバにアクセスしてメモリデータを取得するようにした．システム情報を取得するのにかかった時間を図 7 に示す．この実験では SGX と SMM を用いるオーバヘッドを比較しやすくするために，データ保護のための暗号化や整合性検査は行わなかった．

SGX のみの場合と SGX/SMM なしの場合の取得時間を比較すると，SGX を用いることによるオーバヘッドは 575ms 程度であることが分かる．これは SSdetector における取得時間の 53%を占めている．このオーバヘッドの内訳としては，メモリデータを取得するために呼び出した 1148 回の OCALL やエンクレイヴ内での IDS の実行が挙げられる．一方，SMM のみの場合と SGX/SMM なしの場合の取得時間を比較すると，SMM を用いることによるオーバヘッドは 510ms 程度であることが分かる．これは SSdetector における取得時間の 47%を占めている．このオーバヘッドの内訳としては，システム情報を取得するために発生させた 1148 回の SMI と BIOS 内での SMM プログラムの実行が考えられる．これらのことから，SGX と SMM を用いるオーバヘッドが大きく，SSdetector ではシステム情報の取得に SGX/SMM なしの場合の 5.4 倍の時間がかかることが分かった．

5.3 実機におけるオーバヘッドの見積もり

この実験は VM 内で行ったため，SMI も仮想化されており，実機での性能とは異なる可能性がある．そこで，SMI の呼び出し時間を VM 内とホスト上でそれぞれ測定した．実機に合わせて，VM でも SMI ハンドラでは何も処理を行わないようにした．測定結果を図 8 に示す．実験結果から，SMI 呼び出しは実機のほうが 0.52ms 高速であることが分かった．IDS がシステム情報を取得するために必要な SMI 呼び出しの回数は 1148 回であることから，実機に実装を行った場合，597ms のオーバヘッド削減を見込むことができる．これは SMM のみの場合の取得時間の 84%を占め，SSdetector における取得時間の 55%となる．一方，SMM での実行は実機のほうが低速となる可能性が高いため，実機では SMM プログラムの実行時間は長くなると考えられる．

同様に，VM では SGX も仮想化されているため，実機での性能とは異なる可能性がある．そこで，仮想化の影響を

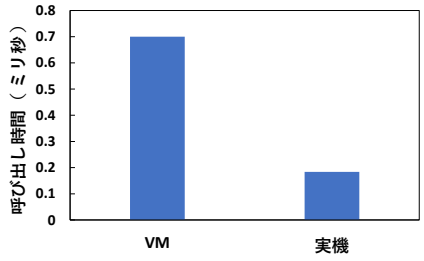

図 8 SMI の呼び出し時間

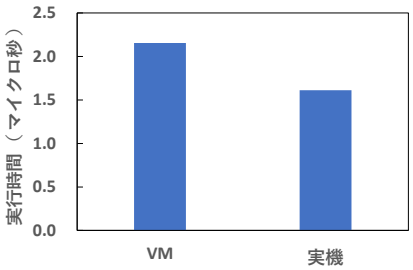

図 9 OCALL の呼び出し時間

最も受けると考えられる OCALL の呼び出し時間を VM 内とホスト上でそれぞれ測定した．この実験では，OCALL 関数で何も処理を行わないようにした．測定結果を図 9 に示す．実験結果から，OCALL 呼び出しは実機のほうが $0.54\mu s$ だけ高速であることが分かる．1148 回の呼び出しでは $0.62ms$ しかオーバヘッドが削減されないため，実機でも SGX のオーバヘッドはほとんど変わらないと考えられる．

5.4 SGX と SMM のオーバヘッドの詳細

さらに詳細に SGX と SMM のオーバヘッドを調べるために，IDS がそれぞれの疑似ファイルを作成するのに必要なシステム情報を取得するのにかかる時間を個別に調べた．各システム情報の取得時間を図 10 に示す．PID/で始まるシステム情報はプロセス番号が PID のプロセスに関する情報取得にかかる時間であり，監視対象システム上で動作していた 286 個のプロセスについての平均となっている．取得したメモリデータはエンクレイヴ内でキャッシュされるため，システム情報を個別に取得するのにかかった時間の合計はまとめて取得した場合と比べて長くなっている．

また，それぞれのシステム情報を取得するために SSdetector が発生させた SMI の回数を図 11 に示す．PID/stat，PID/status，PID/auxv の SMI 呼び出し回数の合計はそれぞれ 888 回，771 回，330 回であった．図 10 と図 11 の測定結果から分かるように，SMI の呼び出し回数とシステム情報の取得時間は比例関係にあり，SMI の呼び出し回数が多いほど取得にかかる時間も長くなる．

システム情報をまとめて取得した場合と異なり，SGX のみの場合より SMM のみの場合のほうが取得時間が長く

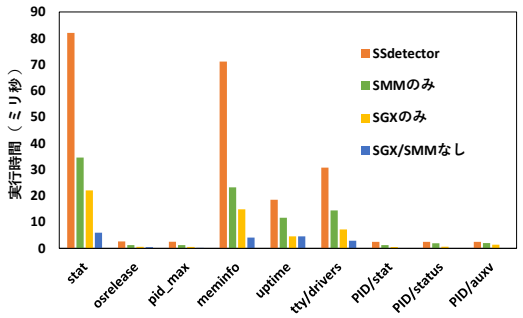

図 10 各システム情報の取得時間

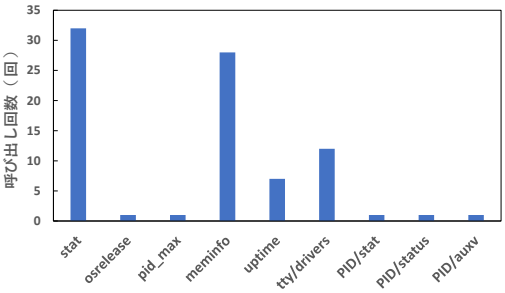

図 11 SMI の呼び出し回数

なっている．つまり，SMM によって生じるオーバヘッドが取得時間に最も大きく影響していることが分かる．また，SSdetector の取得時間に占める SGX と SMM のオーバヘッドが相対的に小さくなっている．これは個別にシステム情報を取得したことによる影響であると考えられる．

5.5 データ保護のオーバヘッド

データ保護のための暗号化や整合性検査によって生じるオーバヘッドを測定するために，暗号化と整合性検査の有無でシステム情報を取得するのにかかる時間を比較した．暗号化と整合性検査を行う SSdetector との比較として，(1) 整合性検査を行わず暗号化のみを行う場合，(2) 暗号化を行わず整合性検査のみを行う場合，(3) 暗号化も整合性検査も行わない場合について測定を行った．測定結果を図 12 に示す．実験結果より，暗号化によるオーバヘッドは 1.75 秒であり，SSdetector における取得時間の 39%を占めていることが分かる．また，整合性検査によるオーバヘッドは 1.71 秒であり，SSdetector における取得時間の 38%を占めている．これらを合計すると SSdetector における取得時間の 77%がデータ保護にかかる時間となり，データ保護を行わない場合の 4.2 倍の時間がかかっている．

データ保護のオーバヘッドは CPU による支援を用いることで削減することができると考えられる．現在のところ，SSdetector の SMM プログラムでは AES-NI 命令セットが利用できていない．これは EDK II が提供している Crypto パッケージの AES 暗号の実装が AES-NI に対応していな

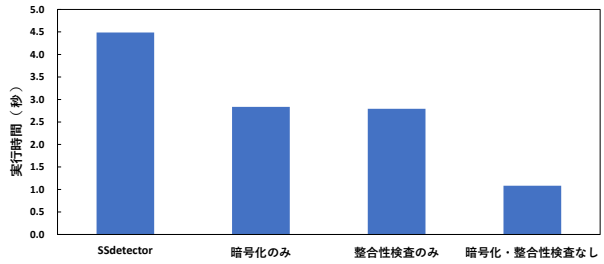

図 12 データ保護のオーバヘッド

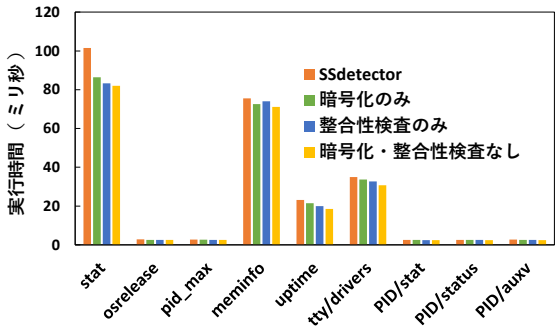

図 13 各システム情報の取得におけるデータ保護のオーバヘッド

いためである．我々の実験によると，AES-NI を用いることにより AES-128-CBC の性能が 3.6 倍に向上したため，暗号化のオーバヘッドは 0.76 秒程度に削減される可能性がある．また，ハッシュ計算の高速化を行う SHA 拡張をサポートした Intel Core i7-12700 を用いて実験を行ったところ，SHA-256 の性能が 3.8 倍に向上した．そのため，整合性検査のオーバヘッドも 0.45 秒程度に削減される可能性がある．その結果，SSdetector における取得時間は 2.29 秒程度になり，データ保護を行わない場合の 2.1 倍程度の時間となる．

さらに詳細にデータ保護のオーバヘッドを調べるために，暗号化と整合性検査の有無でそれぞれのシステム情報の取得にかかる時間を調べた．測定結果を図 13 に示す．SSdetector と暗号化・整合性検査なしの場合の取得時間を比較すると，データ保護のオーバヘッドは最大でも 23%であった．システム情報をまとめて取得した場合にはこのオーバヘッドは 77%であったため，個別に取得する場合にはオーバヘッドが小さくなることが分かる．原因の一つとして，個別にシステム情報を取得する場合にはより時間がかかり，相対的にデータ保護のオーバヘッドが小さくなることが考えられる．

6. 関連研究

SMM を用いたホストベース IDS として HyperGuard [1], HyperCheck [3], HyperSentry [9] がある．HyperGuard は BIOS 内で IDS 全体を実行するため，IDS を更新するたびに BIOS を変更する必要がある．また，SMM による性能への影響が大きい．一方，HyperCheck は SMM でメモリデータの転送のみを行い，リモートホストで監視を行う．そのため，SMM による性能低下を抑えることができるが，リモートホストの IDS の安全性は保証されない．また，NIC ごとに SMM で動作するドライバを開発する必要がある．SSdetector では IDS ごとに SGX アプリケーションを開発すればよく，SMM で動くプログラムは IDS に共通である．

HyperSentry は SMM でハイパーバイザへのエージェントの挿入のみを行う．その後で割り込みを禁止し，１つを除くすべての CPU コアを停止させてエージェントを実行し，安全にシステムの監視を行う．そのため，時間のかかる監視を行うと，割り込み禁止時間やシステムの停止時間が長くなる．SSdetector では SMM でメモリデータを取得している間しかシステムは停止しない．

SGX を用いたホストベース IDS として SGmonitor [2] や SCwatcher [10] がある．SGmonitor はエンクレイヴ内で IDS を実行し，ハイパーバイザを呼び出すことで VM のメモリデータを監視する．また，エンクレイヴ内でファイルシステムを動作させることで VM の仮想ディスクを監視することもできる．しかし，対象が仮想化システムに限定され，ハイパーバイザを信頼する必要がある．SSdetector は SMM プログラムを利用することで，仮想化されていないシステムでも安全に監視を行うことができる．

SCwatcher は，SGX 向け実行環境である SCONE [11] や Occlum [12] を用いて既存の IDS をエンクレイヴ内で実行可能にする．さらに，エンクレイヴ内で proc ファイルシステムを提供することにより，監視対象 VM の情報を既存のインタフェースで取得することができる．このシステムと組み合わせることで，SSdetector でも既存の IDS を動かすことが可能になる．

SGX を用いたネットワークベース IDS には S-NFV [13] や SEC-IDS [14] がある．S-NFV は Snort の内部状態をエンクレイヴ内に格納することによって保護する．エンクレイヴ内に移動した内部状態はエンクレイヴ内のコードのみがアクセスすることができ，外部からの攻撃を受けない安全な API 経由で利用する．これにより，ネットワークフローごとの情報などを盗み見られないようにすることができる．

SEC-IDS は Snort をほとんど修正なしにエンクレイヴ内で実行する．エンクレイヴ内で既存の IDS を動作させるために，Graphene-SGX ライブラリ OS [15] を用いる．また，DPDK を用いることでエンクレイヴ内でネットワークパケットを効率よく取得し，通常の Snort とほぼ同等の性能を達成する．しかし，Snort が取得するまでの間にパケットを書き換えられると，攻撃を正しく検知できない可能性がある．SSdetector では SMM を利用しているため，メモリデータを取得している間に攻撃者が書き換えること

はできない．

SGX と SMM を組み合わせたシステムとして Aurora [16] や Kshot [17] が提案されている．Aurora はエンクレイヴが安全にデバイスを利用することを可能にする．エンクレイヴがクロックやネットワークなどのデバイスにアクセスする際には SMI を発生させて SMM プログラムを呼び出し，SMM でデバイスドライバを実行してデバイスにアクセスする．エンクレイヴと SMM プログラムは共有メモリを経由して暗号化されたデータをやり取りする．これにより，エンクレイヴからデバイスまでの安全な経路が確保される．SSdetector では SMM プログラムを経由してシステムのメモリにアクセスし，監視を行う点が異なる．

KShot は OS やパッチシステムを信頼せずに，カーネルを動かしたままでパッチを適用することを可能にする．エンクレイヴを用いて安全にカーネルパッチをダウンロードして前処理を行い，予約されたメモリに書き込む．そして，SMM プログラムを呼び出し，SMM でカーネルにパッチを適用する．

7. まとめ

本稿では，SGX と SMM を組み合わせることで IDS をより安全かつ柔軟に実行可能にするシステム SSdetector を提案した．SSdetector では SGX のエンクレイヴ内で IDS を実行し，SMM プログラムでシステムのメモリデータの取得のみを行う．エンクレイヴと SMM プログラム間でメモリデータの暗号化および整合性検査を行うことで，IDS が取得するシステム情報の盗聴と改ざんを防ぐ．実験の結果，IDS が監視に用いる proc ファイルシステムに必要なシステム情報を取得できることが確認できた．また，SGX や SMM，暗号化，整合性検査のオーバヘッドは大きいが，改善の可能性があることも示した．

今後の課題は，様々なホストベース IDS を実行できるようにすることである．そのためには，SCwatcher [10] のようにエンクレイヴ内で既存の IDS を動作させられるようにすることが必要である．また，AES-NI や SHA 拡張を用いることで暗号化と整合性検査のオーバヘッドを削減する必要もある．

謝辞

本研究の一部は，JST，CREST，JPMJCR21M4 の支援を受けたものである．また，本研究の一部は，国立研究開発法人情報通信研究機構の委託研究（05501）による成果を含む．

参考文献

[1] J. Rutkowska and R. Wojtczuk. Preventing and Detecting Xen Hypervisor Subversions. *Blackhat Briefings USA*, 2008.

[2] T. Nakano and K. Kourai. Secure Offloading of Intrusion Detection Systems from VMs with Intel SGX. In *Proceedings of the 14th IEEE International Conference on Cloud Computing*, 2021.

[3] J. Wang, A. Stavrou, and A. Ghosh. HyperCheck: A Hardware-Assisted Integrity Monitor. In *Proceedings of International Symposium on Recent Advances in Intrusion Detection*, pp. 158–177, 2010.

[4] EDK II Project. EDK II. `https://github.com/tianocore/edk2`. (Accessed on 06/24/2021).

[5] Intel Corporation. Intel Software Guard Extensions SDK for Linux. `https://01.org/intel-software-guard-extensions/downloads`. (Accessed on 06/24/2021).

[6] Intel Corporation. KVM SGX. `https://github.com/intel/kvm-sgx`. (Accessed on 06/24/2021).

[7] Intel Corporation. QEMU SGX. `https://github.com/intel/qemu-sgx`. (Accessed on 06/24/2021).

[8] Y. Ozaki, S. Kanamoto, H. Yamamoto, and K. Kourai. Detecting System Failures with GPUs and LLVM. In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*, pp. 47–53, 2019.

[9] A. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. Skalsky. HyperSentry: Enabling Stealthy In-context Measurement of Hypervisor Integrity. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, pp. 38–49, 2010.

[10] T. Kawamura and K. Kourai. Secure Offloading of User-level IDS with VM-compatible OS Emulation Layers for Intel SGX. In *Proceedings of the 15th IEEE International Conference on Cloud Computing*, 2022.

[11] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'keeffe, and M. Stillwell. SCONE: Secure linux Containers with Intel SGX. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, pp. 689–703, 2016.

[12] Y. Shen, H. Tian, Y. Chen, K. Chen, R. Wang, Y. Xu, Y. Xia, and S. Yan. Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 955–970, 2020.

[13] M. Shih, M. Kumar, T. Kim, and A. Gavrilovska. S-NFV: Securing NFV States by Using SGX. In *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*, pp. 45–48, 2016.

[14] D. Kuvaiskii, S. Chakrabarti, and M. Vij. Snort Intrusion Detection System with Intel Software Guard Extension (Intel SGX). *arXiv preprint arXiv:1802.00508*, 2018.

[15] C. Tsai, D. Porter, and M. Vij. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *Proceedings of 2017 USENIX Annual Technical Conference*, pp. 645–658, 2017.

[16] H. Liang, M. Li, Y. Chen, L. Jiang, Z. Xie, and T. Yang. Establishing Trusted I/O Paths for SGX Client Systems with Aurora. *IEEE Transactions on Information Forensics and Security*, Vol. 15, pp. 1589–1600, 2019.

[17] L. Zhou, F. Zhang, J. Liao, Z. Ning, J. Xiao, K. Leach, W. Weimer, and G. Wang. KShot: Live Kernel Patching with SMM and SGX. In *Proceedings of the 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 1–13, 2020.