Computer Security Symposium 2018
22 - 25 October 2018

プロセス内処理に対する遠隔認証手法の提案と実装

小松 昌平[1]　PEYROUTAT-BASSE Jerome[1]　瀧本 栄二[2]　毛利 公一[2]　齋藤 彰一[1]

概要：リモート計算機で動くプロセスに対する遠隔認証の必要性が高まっている．静的解析では検出できないデータフロー攻撃を遠隔から検出するために，動的にプロセスの動きをトレースする検出手法がある．この検出手法は規模の大きなプロセスに適用させることは困難である．そこで，脆弱性等の危険性があるコードに限定して検証するために，ユーザーの指定した関数とそこから呼び出される関数群だけをトレースすることができる手法を提案する．そのためのプロトタイプの設計と実装を行い評価を行った．Intel Pin を用いて動的にプロセスの書き換えを行うことで，範囲を指定することを可能にし，トレースの際には Intel SGX を用いることにより，トレース情報の改ざんを防ぐことができる．

キーワード：Remote Attestation，遠隔認証，Intel Pin，Intel SGX，制御フロー

1. はじめに

クラウドの普及に伴い，サーバーをネットワーク経由で管理することが一般化している．さらに，IoT の普及により，IoT デバイスの状態確認の必要性も増加している．このために，遠隔のシステムを状態確認するために Remote Attestation(以下 RA とする) の必要性が高まってきている．この仕組みは，遠隔の計算機上で動くプロセスが本当に正しい動きをしているか，コードが改ざんがなされていないかどうかを認証できる．

基本となる処理を図 1 に示す．認証を要求する側を認証側，要求を受ける側を被認証側という．認証側からの RA 開始のリクエストを受け取ると (1)，被認証側は被認証プロセスバイナリのハッシュ値を計算し (2)，計算結果を認証側に送る (3)．認証側は受け取った値が正しいハッシュ値であるかをデータベースを用いて確認する (4) ことで認証を行う．しかし，このようなバイナリファイルに対する RA 手法には，制御フローを変更する実行時の異常を検知できないという問題が存在する．これは，プロセス自体のコードを変更せずに起こる異常 [1,2] であるため，被認証プロセスバイナリのハッシュ値を確認するだけでは検知ができないために発生する問題である．

実行時の認証問題への対策として，C-FLAT [3] が提案されている．C-FLAT は，被認証プロセスバイナリから計算したハッシュ値ではなく，その対象プログラムの実行時の動きを制御フローによってあらかじめ求め，Basic Block[*1](以下，BB とする) 単位のハッシュ値を確認することで，実行時の異常を検出する．ハッシュ値の計算は実行時の制御フローグラフの各ノードと関連するプログラムコードの開始アドレスとリターンアドレス，前のノード時点でのハッシュ値の三つを用いて行うため，ハッシュ値は実行時のパスによって異なる．このハッシュ値が予め求めた正しいパスを通ったときの値と同じか否かを照合することで実行時の動きが正しいかを認証する．

しかし，被認証プロセスのサイズが大きくなると 2 種類の問題が発生する．一つ目は，処理によって通るパスの数が増加することによる，事前に生成するパス数の増加である．二つ目は，ハッシュ計算による実行速度の低下である．我々は，事前のパスの生成と実行時のハッシュ値計算をプロセスの一部分のみに限定することによって解決することを検討している．その始めとして，本論文では二つ目の実行速度に関する問題に対して，ユーザーが指定したプログラムの一部分に対するパスのハッシュ値による認証について述べる．

本論文では，ユーザーが範囲を指定可能な部分的に RA を行う手法を提案する．これは，C-FLAT では，プロセスの実行前にバイナリを書き換えているため，書き換え範囲は常にプログラム全体であることとは異なる．本手法を利用することにより，ユーザーは大きな被認証プロセスの脆弱性が想定される部分に対してのみ制御フローによる認証手

[1] 名古屋工業大学
Nagoya Institute of Technology
[2] 立命館大学
Ritsumeikan University

[*1] 分岐命令から次に出現する分岐命令までを一つのブロックとみなした単位

© 2018 Information Processing Society of Japan

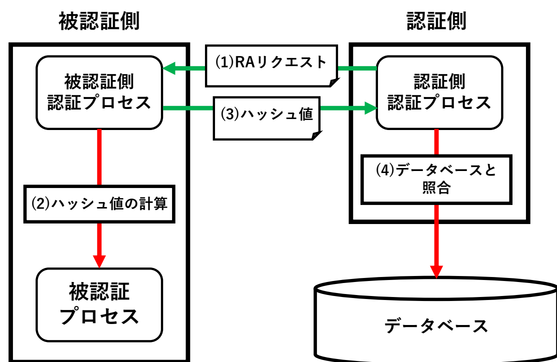

図 1 RA の流れ

法を適用し，実行速度の低下を最低限に抑えつつ効率的な RA をすることができる．また，C-FLAT が計算結果に正当性を持たせるために，ARM Trustzone（以下，Trustzone とする）[4] を用いて行っているハッシュ値の計算と署名を，本提案手法では Intel Software Guard eXtensions(以下，Intel SGX とする) [5] を用いることで，実行時のオーバヘッドを抑える．

以後，本論文では 2 章で関連研究について述べ，3 章で提案手法で用いるハードウェア基盤のセキュリティーシステムである Intel SGX についての説明を行い，4 章で提案手法について述べる．続いて，5 章では提案手法の設計と実装について詳しく述べ，6 章でプロトタイプに対して評価を行う．7 章で今後の課題について述べる．

2. 関連研究

本章では制御フローについての関連研究について述べ，さらに，セキュリティ技術である ARM TrustZone について述べる．次に，TrustZone を使用して制御フローに基づく RA を実現している C-FLAT について，まず概要について述べ，ハッシュ計算，バイナリ書き換えについて述べる．

2.1 制御フローに関する関連研究

プロセスの制御フローの完全性に関する研究 [6] がある．この研究では，プロセスの動きが正しいものであるかどうかを知るために，制御フローを確認することに対する完全性を述べたものである．この論文から，実行時の制御フローを RA することでリモートからプロセスを認証の認証が可能であることを示している．

2.2 ARM TrustZone

TrustZone は ARM アーキテクチャにおけるセキュリティ技術であり，実行環境を「Normal World」と「Secure World」の二つに分けることができる．Normal World から Secure World へのアクセスは制限されており特定の API を経由する必要があるため，Secure World 内のメモリ上に存在するデータに対して Normal World 内のプロセスがアクセスすることはできない．また，Secure World と Normal World の切り替えはコンテキストスイッチで行われるため，レジスタ状態の保存や取り出しによって切り替えのオーバヘッドが生じる．

2.3 C-FLAT

C-FLAT は IoT 機器などにおける組み込みソフトウェアを対象にした RA を実現している．C-FLAT は被認証プロセスバイナリのハッシュ値を RA するのではなく，被認証プロセスの動きを BB 単位で実行履歴と実行中アドレスのハッシュ値に基づいて RA を行うことで，プログラムコードを変更しない制御フローを変更する攻撃を検出することができる．検出のために，実行前に被認証プロセスバイナリに対して，BB 単位でハッシュ値を計算するコードの追加を行う．また，TrustZone を用いることで，ハッシュ計算と計算結果への署名を Secure World の中で行うことでハッシュ計算結果の改ざんを防ぎ，計算結果に正当性を持たせている．

2.3.1 バイナリ書き換え

実行前に，Capstone disassembly engine [7] というツールを用いて予めバイナリに対して書き換えを行う．ARM アーキテクチャのリンク付きの分岐命令である bl 命令をフックし，分岐先の開始アドレスと，分岐先からのリターンアドレスの値をノード情報として Secure World 内に渡すための書き換えを行う．このようにすることで，BB の遷移毎に実行中のアドレス情報に基づいてハッシュ計算を行うことができる．また，プログラムの書き換えは実行前に行い，実行中に書き換えることはしない．

2.3.2 ハッシュ計算

ハッシュの計算は，制御フローのグラフにおける各ノード単位で行い，随時更新を行う．この計算は Secure World 内で行うため，計算結果を任意のプロセスによって書き換えられることはない．あるノード n+1 におけるハッシュ値 H_{n+1} は (1) の式で計算される．I_n は n 番目のノード情報であり，C-FLAT においては各ノードに紐づくプログラムコードの先頭アドレスとリターン先アドレスである．

$$H_{n+1} = Hash(I_n, H_n) \tag{1}$$

ハッシュ計算の一例を図 2 に示す．パス [N1 → N2 → N4] では，ノード N2 で求めるハッシュ値は前ノード N1 のハッシュ値である H_1 と N2 のノード情報（遷移先のノードに関連するコードの先頭アドレスと遷移先からのリターンアドレス）である I_2 を用いてハッシュ計算を行う．ノード N4 においても同様に前のノードのハッシュ値 H_2 を用いて計算を行う．一方，パス [N1 → N3 → N4] についても同様の計算方法で計算を行うので，ノード N4 におけるハッシュ値 H_4 はそれぞれのパスで異なったものになる．これ

© 2018 Information Processing Society of Japan

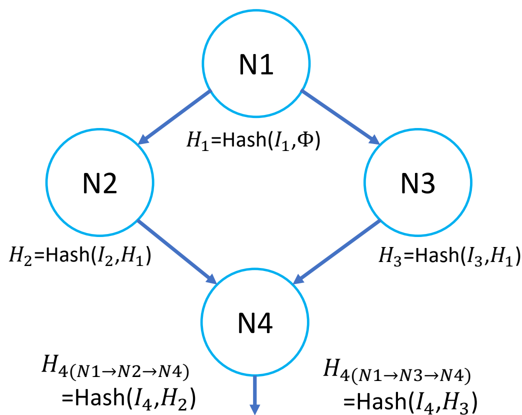

図 2 ハッシュ計算方法の一例

により，どのパスを通ったのかをハッシュ値から特定することができる．

3. Intel SGX

本章では，提案手法において用いる Intel SGX について述べる．

3.1 概要

Intel SGX は Intel CPU の命令セットの一つでありプロセスがメモリ上に Encalve と呼ばれる領域を作成することができる．Enclave はメモリ上では暗号化されており，実行時には CPU 内で復号されるため，平文がメモリ上にロードされることはない．

3.2 Enclave

Enclave にアクセスできるのは，当該 Enclave を作成したプロセスのみである．作成したプロセス以外のプロセスや OS，他の Enclave 領域からはアクセスできない．ただし，他の Enclave 領域からのアクセスは，互いの Enclave へのアクセス鍵の交換を完了することで可能になる．

Enclave を作成するためには，sgx_create_enclave という API を呼び出し，アクセスをするためには sgx_ecall と呼ばれる API を呼び出し Enclave 内の関数やデータにアクセスする．

3.3 TrustZone との違い

3.2 節で述べた通り，Enclave は作成したプロセス以外，他のプロセスや OS，また他の Enclave からもアクセスすることができない．そのため，プロセス毎の秘匿処理などを OS や他のプロセスから保護することができる．対して TrustZone は，Secure World 内の領域に対して他の Secure World 内のプロセスや OS はアクセスすることが可能であるという点が大きく異なる．また，TrustZone はコンテキストスイッチによって Secure World と Normal World 間の状態遷移を行うのに対して，Intel SGX は CPU 内での復号のみであるため，呼び出しコストは Intel SGX の方が小さい．

4. 提案手法

本章では提案手法についての概要を述べ，その後データベース作成フェーズ，実行フェーズ，RA フェーズそれぞれについて述べる．

4.1 概要

本提案手法では，被認証プロセスの制御フローに基づく RA を，ユーザーが指定したプロセス内の一部分の処理に対しての適用する手法を提案する．C-FLAT のような制御フローに基づくプロセス認証を大きなプロセスなどに適用させる場合，二つの問題がある．一つ目は制御フローグラフの大きさが大きくなり，その作成が困難になる点である．二つ目は，制御フローをトレースするコストの増加とそれによる実行速度の低下である．このため，C-FLAT では IoT などの比較的小さなプロセスを対象としている．

本提案手法では，これらの問題のうちの実行速度低下の問題に対応するため，ユーザーの指定する一部分の処理に対してのみトレースを行う手法を提案する．さらに，トレース範囲を固定するのではなく実行中にユーザーの指定を基に被認証プロセスのバイナリを書き換えることによって，動的にトレース範囲を変更可能とする．これにより，必要なタイミングで必要な範囲に対して制御フローによる RA を実現して，実行速度の低下を抑制する．なお，本手法は，事前に制御フローグラフを作成するコストは変わらない．

また，C-FLAT が TrustZone を用いているのに対して，本提案手法では Intel SGX を用いてハッシュの計算，計算結果への署名を行うことで，実行時オーバヘッドの削減をする．

4.2 データベース作成フェーズ

実行前に，正常状態となる制御フローグラフを定義するためにあらかじめ対象のプロセスに対して解析を行い，ハッシュ値の計算を行う．この際，すべての実行パスについてハッシュ値の計算を行う．計算結果はデータベースに保存し，RA フェーズにおいてこの作成したデータベースを用いて計算結果の照合を行う．

4.3 実行フェーズ

プロセスがユーザーからのリクエストを受信するまではハッシュの計算は行わない．ユーザーである認証側からプロセス内の指定部分をリクエストとして受け取ると，被認証側はその指定部分に対して被認証プロセスバイナリの書き換えを始める．その後，書き換えながらプロセスを実行し実行時の動きをトレースする．プロセスの実行がトレー

© 2018 Information Processing Society of Japan

スの基本単位（BB や関数）から別の単位に遷移するたびに SGX 関数を call し Enclave に入りハッシュの計算を行う．ハッシュ計算が不必要な部分の計算を行わないようにすることで，速度の低下を抑えることができる．

4.4 RA フェーズ

認証側からの RA のリクエストを被認証側が受け取ると，Enclave 内において計算結果に対して署名を行い計算結果を認証側に送信する．認証側は，受け取った計算結果の署名を確認しあらかじめ作成してあるデータベースと計算結果の照合を行う．

4.5 部分的な認証によるメリット，デメリット

部分的な認証によるメリットとしては，認証プロセスが大きな場合であっても制御フローの RA が行えることが挙げられる．例えば，被認証プロセスの中で脆弱性がある可能性のある部分のみに限定してトレースすることで，全体的なシステムの適用時の実行時間を減らすことができることが挙げられる．または，システム監査時のみに RA を有効化することによって，日常の実行速度の低下を防ぐことができる．このように，必要なときに必要な部分のみを監査できることは大きなメリットがある．なお当然ながら，指定を行った部分以外に異常があった場合は検出することができないことはデメリットである．

5. プロトタイプの実装

本章では，作成したプロトタイプシステムの実装と，実装で用いるツールである Intel Pin [8](以下，Pin とする) について述べる．なお，本プロトタイプはトレースの単位として BB ではなく関数を使用している．トレースの粒度は低下するが，基本機能の確認には十分であると考える．

5.1 システム全体構成

図 3 を用いて全体の構成を述べる．図 3 の RA ライブラリとは，Enclave 内で行うハッシュ計算や計算結果への署名，RA の処理を行う自作の共有ライブラリである．RA ライブラリ内の RA スレッドがリクエストを受け取ると (1)，RA スレッドは Pin ライブラリに受け取った範囲のリクエストを伝える (2)．Pin 内で実行時の命令挿入を行う Pin ライブラリは，受け取った範囲から被認証プロセスの書換範囲の指定を行う (3)．

実行中には，Pin 内で call 命令または ret 命令をフックし (4)，RA ライブラリのハッシュ計算用のスレッドの関数を呼び出す (5)．ハッシュ計算スレッドは Enclave に入りハッシュ計算を行い，ハッシュ値の更新をする (6)．

また，RA スレッドは開始のリクエストを受け取ると (7)，Enclave に入りその時点でのハッシュ値に対して署名を行い，その結果を認証側に送信する (8)．

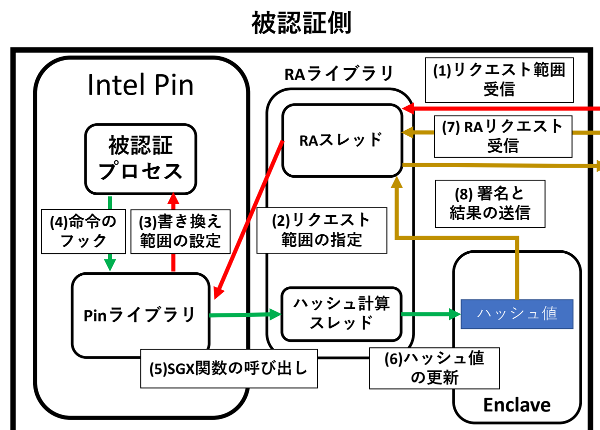

図 3 被認証側システム構成図

5.2 データベースの作成

データベースの作成のために，Dyninst [9] を用いて解析を行う．Dyninst はバイナリ解析ツールであり，静的な解析機能や動的に命令を挿入するための API を提供している．静的な解析によって取得した被認証プロセスのフローグラフから，すべてのパスを抽出する．すべてのパスについて開始アドレスとリターンアドレスに基づくハッシュ値の計算を行い，計算結果をデータベースに保存する．

5.3 被認証バイナリの書き換え

本節ではバイナリの編集について述べる．その際に用いる動的編集ツールである Pin について述べ，実行フェーズの実装について述べる．

5.3.1 Intel Pin

Pin は DBI(dynamic binary instrumentation) フレームワークであり，動的なバイナリコードの解析と編集を行うことができる．Pin を使用することで，実行した命令の取得，各レジスタの値の取得，新規コードの挿入などを行うことができる．この解析及び編集は，関数，命令などの単位で行う．本提案手法において Pin は大きく分けて SGX 関数の挿入とユーザーリクエストに基づいた部分的なバイナリ書き換えの二つの処理をするために用いる．

本論文においては，プロトタイプの設計であるため，BB 単位ではなく関数単位での大まかな遷移をトレースする．そのため，バイナリの書き換えは，ユーザーが指定した範囲内の call 命令のフックとなる．また，バッファオーバフロー検知のために ret 命令も同様にフックしてトレースを行う．フック先で，SGX の Encalve を呼び出すための関数を挿入して，ハッシュ計算を行う．

5.3.2 Pin によるコード挿入

Pin の実行時の命令挿入は Pin ライブラリ内で行う．例を図 4 に示す．8 行目の INS_AddInstrumentFunction で，実行したすべての命令に対して，2 行目から始まる Instruc-

© 2018 Information Processing Society of Japan

```
1 VOID docount() { icount++; }
2 VOID Instruction(INS ins, VOID *v)
3 {
4     INS_InsertCall(ins, IPOINT_BEFORE,
5         (AFUNPTR)docount, IARG_END);
6 }
7 int main(int argc, char * argv[])
8 {
9     if (PIN_Init(argc, argv))
10    return Usage();
11
12    INS_AddInstrumentFunction(Instruction, 0);
13
14    PIN_StartProgram();
15 }
```

図 4 Pin ライブラリサンプルコード

tion 関数を呼びだして 4 行目の INS_InsertCall を用いて関数 docount を挿入する．INS_InsertCall 関数は，第二引数でその命令の前に挿入するか後に挿入するかを指定し，第三引数には挿入する関数のアドレスを指定する．図 4 のサンプルコードでは，すべての実行命令の前に関数 docount を挿入し，実行した命令の数をカウントする．命令挿入の終了後，13 行目の PIN_StartProgram でプログラムをスタートする．

書き換えは，コードを初めて実行するときにのみ行われる．以降，当該プロセスが生きている間は Pin 内のキャッシュが有効なため，実速度での実行が可能である．しかし，キャッシュは実行毎に破棄されるため，何度も被認証プロセスを実行する場合は毎回書き換えが必要である．また，書き替えたコードをファイルにダンプすることも出来ない．

5.3.3　指定部分に対するバイナリ書き換え

ユーザーから受け取ったリクエストを元に，Pin 内で被認証プロセスバイナリの書き換えを行う．まず，書き換えは実行中のプロセスを動作させたままであることが望ましい．しかし，Pin の制約により，一旦 Pin とプロセスを再起動して書き換えを開始する．なお，Web サーバーなどの多数のセッションを扱うようなプロセスの場合は，プロセスの継続性に問題はないと思われる．なお，この再起動問題については検討を続ける予定である．

関数名でリクエストを受け取った時の範囲指定の例を図 5 を用いて説明する．Pin は IMG と呼ばれる単位で実行可能ファイルを管理する．認証側から関数 func1 をトレースするリクエストを受け取ると，IMG 毎の処理関数 Image 内で，RTN_FindByName 関数 (5 行目) を呼び出す．RTN は Pin 内における関数の単位であり，RTN_FindByName 関数で被認証バイナリの IMG である binary_img から func1 という名前の RTN を探す．その後，7 行目から 11 行目で，発見した RTN から func1 の先頭アドレスと終了アドレスを取得する．8 行目の RTN_Address 関数は，RTN の先頭アドレスを取得する関数である．また，9 行目の終了アドレスは，先頭アドレスに RTN のサイズを足して算出しており，RTN のサイズの取得には RTN_Size 関数を用いている．これにより，目的の関数 func1 のアドレス範囲を取得

```
1 VOID Image(IMG binary_img, VOID *v)
2 {
3    const char *func_Name ="func1";
4    RTN function_rtn
5     = RTN_FindByName(binary_img,func_Name);
6
7    ADDRINT trace_start_address
8     = RTN_Address(function_rtn);
9    ADDRINT trace_end_address
10    = trace_start_address
11      +RTN_Size(function_rtn);
12 }
13 int main()
14 {
15   IMG_AddInstrumentFunction(Image,0);
16   PIN_StartProgram();
17 }
```

図 5　アドレス範囲の指定

することができる．

加えて，実行命令毎の処理関数 Instruction の中での処理を図 6 を用いて説明する．IMG の処理で取得した認証側指定範囲内の call 命令と ret 命令だけではなく，その範囲内から呼び出された先の関数内での処理もトレースする必要がある．そのため，1 行目で宣言している flag を用いて，flag が true の時は呼び出された先の関数内の処理をトレース中，false の時は呼び出された関数先以外のトレース中である．7 行目から 21 行目は flag が true である場合の処理である．8 行目の INS_IsCall 関数は，実行された命令 ins が call 命令かどうかを判別する関数である．もし，call 命令であった場合は SGX 関数を挿入する．また，11 行目の INS_IsRet 命令は引数が ret 命令かどうかを判別する．ret 命令であった場合は 12 行目で ret 先 (BRANCH_TARGET_ADDR) が範囲内であるかを range 関数でチェックし，範囲内であれば範囲内トレースに状態が変わるので flag を false にして，SGX 関数を挿入し，範囲外であれば SGX 関数の挿入だけを行う．22 行目から 33 行目では，flag が false の時かつ指定範囲内の命令に対する処理を示している．22 行目で，実行した命令のアドレス ins_add が範囲内であるかをチェックし，call 命令であれば呼び出した先の関数内の処理のトレースに状態が変わるので，SGX 関数を挿入して flag を true にする．29 行目では，範囲内の ret 命令の場合 SGX 関数の挿入のみを行う処理を示している．

5.3.4　Pin における SGX の利用

Pin は仕様上独自のライブラリ管理を行っており，動的にライブラリをロードできない．そのため，SGX の Enclave を使用するための標準ライブラリである SGX ライブラリと RA ライブラリを共有ライブラリとしてコンパイルし，IMG として Pin に認識させている．なお，SGX の API を利用する場合は，Pin ライブラリ内で RTN_FindByName を用いて SGX ライブラリ内の関数を探して利用している．

RA ライブラリ内でハッシュ計算スレッドと RA スレッ

© 2018 Information Processing Society of Japan

```
bool flag=false;
VOID SGX_FUNC(){
//call Enclave and hash calculation
}
VOID Instruction(INS ins, VOID *v)
{
  if(flag==true){
    if (INS_IsCall(ins)){
      INS_InsertCall(ins, IPOINT_BEFORE,
                     (AFUNPTR)SGX_FUNC, IARG_END);
    }else if (INS_IsRet(ins)){
      if(range(BRANCH_TARGET_ADDR)){
        INS_InsertCall(ins, IPOINT_BEFORE,
                       (AFUNPTR)SGX_FUNC, IARG_END);
        flag = false;
      }
      else{
        INS_InsertCall(ins, IPOINT_BEFORE,
                       (AFUNPTR)SGX_FUNC, IARG_END);
      }
    }
  }else if (ins_add >= range_start_address
          && ins_add <= range_end_address){
    if (INS_IsCall(ins)){
      INS_InsertCall(ins, IPOINT_BEFORE,
                     (AFUNPTR)SGX_FUNC, IARG_END);
      flag = true;
    }
    if (INS_IsRet(ins)){
      INS_InsertCall(ins, IPOINT_BEFORE,
                     (AFUNPTR)SGX_FUNC, IARG_END);
    }
  }
}
int main()
{
  INS_AddInstrumentFunction(Instruction,0);
  PIN_StartProgram();
}
```

図 6 実行命令トレース方法の指定

ドの二つが動作しており，トレース時にはハッシュ計算スレッド内の関数を，call 命令と ret 命令の直前に入れることで Enclave に入りハッシュの計算を行うことができる．提案手法では，ハッシュ計算に用いる情報を call 命令であれば call 先のアドレス，ret 命令であれば ret 先のアドレスとしている．Encalve 内のハッシュ値は計算毎に更新されて常に最新の値が保存されている．

5.4 RA フェーズ

被認証側は RA のリクエストを受け取ると，その時点での Enclave 内のハッシュ値に対して署名を行い，認証側に送信する．通信は Google Protobuf [10] を用いて実装を行なっている．ハッシュ値を受け取った認証側はデータベースにアクセスし照合を行うことで，RA が完了する．

6. 評価

本提案のシステムのプロトタイプを用いて，評価を行った．本章では，それぞれの評価項目に対する結果について述べる．評価環境を表 1 に示し，評価項目を以下に挙げる．

- パスの異常検知
- システム適用時のオーバヘッド
- オーバヘッド内訳

表 1 評価環境

OS	Ubuntu 16.04
CPU	Intel Xeon E3-1245 3.5GHz
SGX SDK	Version 2.6.0
Intel Pin	Version 3.5

```
void func1(){
  char buf[8];
  fgets(buf, 100, stdin);
  return;
}
int main()
  func1();
  return 0;
}
```

図 7 評価用被認証プロセスコード

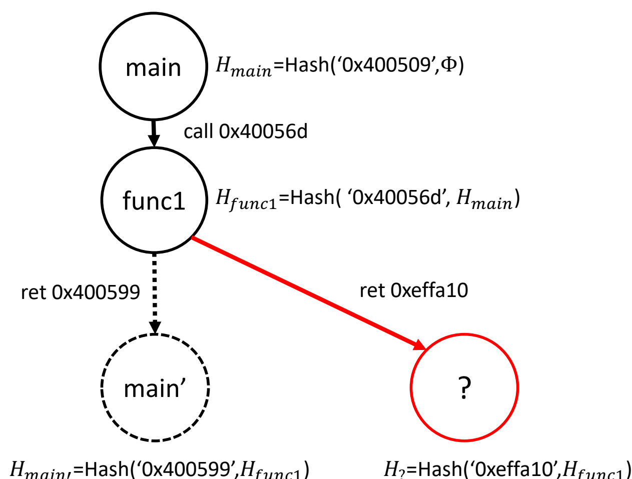

図 8 ハッシュ値による異常検知

パスの異常検知は，実行パスを変更する攻撃としてスタックオーバーフロー攻撃を行った．システム適用時のオーバヘッドは，システム未適用時の実行時間とシステム適用時の実行時間を比較する．また，オーバヘッド内訳はオーバヘッドの詳しい内訳を調査する．

6.1 パスの異常検知

評価に用いた被認証プロセスのコードを図 7 に示す．被認証プロセスに対して，実行パスを変更する攻撃としてバッファオーバーフロー攻撃を行い，正しく検知ができることを確認した．その時のノード遷移を図 8 に示す．func1 から main へのリターンアドレスが攻撃により書き換えられ，意図しないノードに遷移した場合を赤い線で示している．スタックオーバーフロー攻撃により，ret 先のアドレスは正しいアドレスとは異なる値になり，最終的なハッシュ値の値もデータベースに格納している値とは一致しないため，検出をすることができる．

6.2 システム適用時のオーバヘッド

オーバヘッドの比較結果を表 2 に示す．この表における

© 2018 Information Processing Society of Japan

表 2 実行時間

event 回数 (回)	10^2	10^3	10^4	10^5	10^6
未適用時 (msec)	0.001	0.007	0.066	0.692	4.432
適用時 (msec)	10.0	43.5	342.3	3478	38169

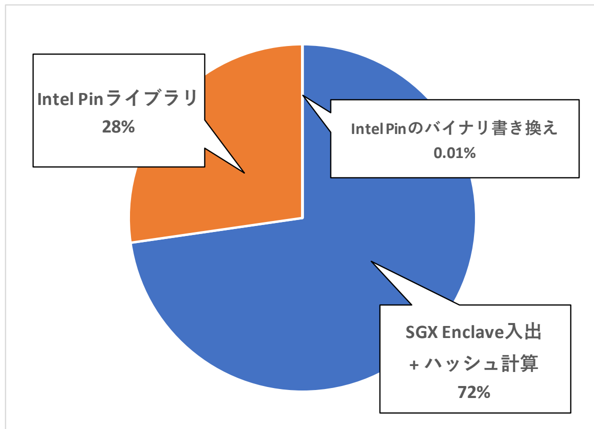

図 9 適用時オーバヘッド内訳

イベントとは，call または ret 命令の実行によって，Enclave に入りハッシュ計算を行うことである．この結果より，システム適用時は未適用時の実行時間に対して約 500 倍程度の時間が必要となることが分かる．しかし，この評価における未適用時の被認証プロセスは中身が空である関数をループで call するというものであるため，関数内での処理が存在する一般的な被認証プロセスの場合は，この結果よりも実行時間の差が小さくなると考えられる．

6.3 オーバヘッド内訳

システム適用時にかかるオーバヘッドを測定した．その割合を表したものを図 9 に示す．オーバヘッドのうち，七割以上を SGX の Enclave への出入りとハッシュ値の計算が占めていることがわかる．これはイベント毎に呼び出されるためイベント回数に応じてオーバヘッドが増加する．

また，三割弱を占めている Pin ライブラリによるオーバヘッドは，ライブラリ内で使われる PIN_CallApplicationFunction() という関数によって発生している．この関数は，Pin ライブラリ内において被認証プロセスバイナリに対して，他の IMG 内の関数を挿入する際に用いられる関数で，オーバヘッドが大きい関数である．しかし，本提案手法の実装のため SGX の Encalve を呼び出し，Enclave 内でハッシュ計算を行う関数を挿入する必要があるため，この関数を使用することが必要である．

7. 今後の課題

実行速度向上のための今後の課題として，まず一つ目にオーバヘッドの削減が挙げられる．そのためには，ハッシュ値の計算回数を削減する必要がある．例えば，同じ関数が何度も続けて call されているときは，最終的に呼ばれた回数をハッシュ値の計算に含めることで，何度もハッシュ値の計算を行う必要がないようにすることでオーバヘッドの削減ができると考える．

二つ目は，ループ時のハッシュ計算の削減についてである．本プロトタイプでは，ループを分けて処理していない．しかし，ループにおいては，ループの回数やループ内でのパスについてもハッシュ値の結果に加える必要がある．そのため，ループは，全体のパスとは独立して生成をする必要がある．その実現には，データベース作成フェーズの際に，ループの開始と終了の場所を見つける必要がある．実行時に被認証側はその情報を用いて，ループの回数とループ内でのパスのハッシュ値を全体のパスに情報として加えることで，実行時の動きをより正確に計測することができると考える．

8. おわりに

本論文では，制御フローによる RA をユーザーの指定したプロセス内の一部分の処理に対しての適用するための手法を提案し，プロトタイプの実装と評価を行った．本提案手法は，部分指定を可能にすることで，制御フローによる RA による速度低下がある大きな被認証プロセスに対して適用することが可能である．今後は，オーバヘッド削減のためのハッシュ計算方法の変更，ループへの対応を行う．さらに，全体パス増加への対応を検討する．

参考文献

[1] Marco Prandini, M. R.: Return-Oriented Programming, *IEEE Security & Privacy*, Vol. 10 (2012).

[2] T. Bletsch, X. Jiang, V. W. F. and Z.Liang: Jump-Oriented Programming: A New Class of Code-Reuse Attack, *ACM ASIACCS* (2011).

[3] Tigist Abera, N.Asokan, L. D. J.-E. E. T. N. A. P. A.-R. S. G. T.: C-FLAT: control-flow attestation for embedded systems software, *Computer and Communications Security*, Vol. 2016, pp. 743–754 (2016).

[4] Alves T, D.: TrustZone: Integrated Hardware and Software Security (2004).

[5] Intel, C.: Intel SGX Homepage, `https://software.intel.com/en-us/sgx`.

[6] Martn Abadi, Mihai Budiu, c. E.-J. L.: Control-flow integrity principles, implementations, and applications, *ACM Transactions on Information and System Security (TISSEC)*, Vol. 13, No. 4 (2009).

[7] Capstone: Capstone The Ultimate Disassembler, `http://www.capstone-engine.org/`.

[8] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J. and Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation, *ACM SIGPLAN Notices*, Vol. 40, No. 6, pp. 190–200 (2005).

[9] Paradyn, Tools, P.: Dyninst Putting the Performance in High Performance Computing, `https://dyninst.org/`.

[10] Google, C.: Google Developers:Google Protocol Buffers, `https://developers.google.com/protocol-buffers/`.

© 2018 Information Processing Society of Japan