Vol.2020-DPS-182 No.27
Vol.2020-CSEC-88 No.27
2020/3/12

IoTデバイス向けの制御フローベース遠隔認証手法の軽量化の検討

吉野 貴史[1]　掛井 将平[1]　瀧本 栄二[2]　毛利 公一[2]　齋藤 彰一[1]

概要：IoT 技術の普及と IoT デバイスに対する攻撃の増加に伴い，IoT デバイスの動作を認証する必要性が高まっている．デバイスの動作を認証する既存手法としてデバイス上のプロセスの制御フロー情報から遠隔認証を行う手法がある．しかしこの手法は制御フロー情報の取得と計算のコストが大きく，低リソースな IoT デバイスには不適である．そこで本研究では制御フローベースの遠隔認証技術を IoT デバイスへ適用する手法を提案する．計算コストが大きい処理を IoT デバイスを管理する IoT ゲートウェイ上で行い，デバイス上の動作コストを削減し，IoT デバイスへの適用を目指す．

キーワード：IoT，Remote Attestation，Intel SGX，制御フロー

1. はじめに

IoT(Internet of Things) 技術の普及に伴い，これまでインターネットに接続されていなかった家電やセンサーなどがインターネットに接続している．IoT デバイスの中には CPU やメモリのリソースが少なく，十分なセキュリティ機能を持たない脆弱な機器が存在するために，脆弱な機器を狙った攻撃が増加している．2016 年には「Mirai」[1] とよばれるマルウェアによって脆弱性を持つ IoT 機器が多数乗っ取られ，当時最大規模の DDoS 攻撃が行われた．このことからも，IoT デバイスのセキュリティ機能の向上は急務であるといえる．

デバイスの保護手法として遠隔認証 (Remote Attestation) を用いてデバイスの状態を監視する手法 [2] がある．Remote Attestation(以下 RA という) は遠隔の計算機上で動作するプロセスが正常に動作しているか，もしくは改ざんされたものではないかを認証する技術である．RA は通常，認証を行う認証者と，認証の対象となるデバイスとの間で行われる．認証者はデバイスを認証するために，デバイスの完全性を示す認証データを用いる．この認証データをデバイスのどのような情報から生成するかによって，認証者が認証可能な情報が変化する．例として認証データにデバイス上で実行されるコードのハッシュ値を用いた手法について述べる．この手法では認証者は正常なコードのハッシュ値と，デバイス上のコードのハッシュ値を比較し，デバイス上のコードが改ざんを受けたものではないかを認証する．しかし，この手法では認証者は認証を行ったコードの実行内容に関して情報を得ることができない．そのためコード変更を伴わない攻撃 [3] の検知には対応していない．

RA を用いたデバイス認証の研究に C-FLAT(Control-Flow Attestation for Embedded Systems Software)[4] がある．C-FLAT はプログラムの全体の制御フローを表した Control Flow Graph(CFG) と実行時の制御フローを用いて RA を行う研究である．実行の分岐処理毎にそれまでの処理の流れを表す値を求め，この値を制御フロー情報としてプロセスの制御フローの認証に用いる．これにより，制御フローを変更する攻撃を検知することが可能となる．しかし，C-FLAT には認証データとして用いる制御フロー情報の計算コストが大きいという問題がある．このため被認証プロセスが動作するデバイス上に十分なリソースが必要になり，低リソースの IoT デバイスには適用が困難である．

本論文では IoT デバイスに適した形での制御フローベースの RA システムを提案する．対象のデバイスは低リソースの IoT デバイスとし，RA には C-FLAT で用いられる手法を用いる．問題点であった認証データの計算は，IoT デバイスを管理する IoT ゲートウェイ上で行うことでデバイス上の動作コストを削減し，IoT デバイスへの適用を目指す．また計算の正当性の確保のため Trusted Execution Environment(TEE) 技術の 1 つである，Intel SGX[5] を用いる．

以後，本論文では 2 章で TEE について述べたあと，3 章

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

で C-FLAT の詳細と問題点を関連研究を踏まえて述べる．続いて 4 章で提案手法について述べたあと，5 章で実装について詳しく述べ，6 章で評価を述べ，7 章で今後の課題について述べる．

2. Trusted Execution Environment

本章では本提案や関連研究の C-FLAT で用いられる TEE について述べる．初めに C-FLAT で用いられる ARM アーキテクチャが持つ ARM TrustZone[6] について述べ，次に提案で用いる Intel SGX について述べる．

2.1 ARM TrustZone

TrustZone は ARM アーキテクチャが持つセキュリティ技術である．TrustZone では実行環境が「Normal World」と「Secure World」に分かれている．Normal World は通常の実行環境である．Secure World は通常の実行環境から隔離された実行環境であり，セキュア OS が動作する安全性が高い環境である．Normal World から Secure World へのアクセスは制限されており，SecureWorld の機能の利用には特定の API を経由する必要がある．そのため Secure World のメモリ上のデータに対し Normal World からはアクセスすることができない．このように Secure World 内のデータの保護を行う技術である．

2.2 Intel SGX

Intel SGX は，メモリ上に Enclave と呼ばれる保護領域を作成する．メモリ上の Enclave 領域内のデータは暗号化されており，実行時には CPU 内で復号後実行される．そのため実行時にメモリ上に平文でデータが置かれることなく，他のプロセスからの盗聴を防ぐことが可能となる．

2.2.1 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 領域からのアクセスは不可能である．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．また Enclave を作成したプロセスであっても Enclave へのアクセスは専用 API を通してしか行うことができない．

2.2.2 TrustZone との違い

Intel SGX と ARM TrustZone は保護領域の作成によってセキュアな実行環境を提供する技術であるが，それらの間にはいくつか違いがある．一つは保護領域の性質である．Intel SGX の Enclave には 2.2.1 で述べたように作成したプロセス以外のプロセスや OS，他の Enclave からアクセスできないという特徴がある．それに対して TrustZone は，Secure World 内の領域に対して，Secure World 内の他のプロセスやセキュア OS はアクセス可能である．また呼び出し時にかかるコストにも違いがある．Intel SGX は保護領域内の処理を実行する際に，Enclave 内の情報を CPU で復号し実行を行う．対して TrustZone はコンテクストスイッチによって保護領域である Secure World と通常領域である Normal World 間の状態遷移を行う必要がある．この状態遷移を伴わない分，Intel SGX は呼び出し時のコストが小さい．

ソースコード

```
int main(void)
{
 ①  ⋮
   if(i > j)
  ②    i++;
    else
  ③    i--;
 ④  return 0;
}
```

```
00000000004004d6 <main>:
    ⋮
① 4004dd: cmp    -0x4(%rbp),%eax
  4004e0: jle    4004e8 <main+0x12>

② 4004e2: addl   $0x1,-0x8(%rbp)
  4004e6: jmp    4004ec <main+0x16>

③ 4004e8: subl   $0x1,-0x8(%rbp)

  4004ec: mov    $0x0,%eax
④ 4004f1: pop    %rbp
  4004f2: retq
```

図 1　分岐による BB 区分

3. C-FLAT

本章では C-FLAT の RA 手法についての詳細を述べる．初めに C-FLAT の認証概要と認証に用いるデータの詳細について述べたあと，バイナリ書き換えを用いた制御フロー情報の取得方法について述べ，認証に用いる CFG のハッシュ計算方法について述べる．最後に C-FLAT の提案手法を IoT デバイスに適用する際に発生する問題点について述べ，問題解決を目指した関連研究を述べる．

3.1 認証概要

C-FLAT は被認証プロセスの実行時の Basic Block(BB)[*1] 単位の制御フローとプログラム全体の CFG を元に RA を行う研究である．C-FLAT の認証は Control Flow Integrity(CFI)[7] がベースとなっている．CFI とはプロセスの制御フローを確認することでプロセスの完全性を証明する研究である．この研究から，被認証プロセス実行時の制御フローを元に RA を行うことで，リモートからプロセスの認証が可能であることを示している．

次に C-FLAT の認証で用いる BB 単位の制御フロー情報について述べる．BB 単位の制御フローとは被認証プロセスの実行の流れを BB の遷移で表したものである．単純な分岐のアセンブリコード例を図 1 に示す．アセンブリコードは BB ごとに囲まれ区分され，元となったソースコードはコード左の番号に対応している．コードは 4 つの BB を持ち，BB1 と BB2 は BB1 の条件付きジャンプ命令 (jle) によって分割され，BB3 と BB4 は BB2 のジャンプ命令 (jmp) によって分割されている．図 1 では実行される BB 遷移として [1 → 2 → 4] と [1 → 3 → 4] がある．この BB 遷移の単位で被認証プロセスの実行時の制御

[*1] ブロックは分岐命令やジャンプ命令によって分割される連続した命令列である

フローを取得し，正常な制御フローと比較し認証を行う．また C-FLAT では BB を識別するための情報として各 BB の先頭と末尾のアドレスを用いる．このアドレスが制御フローを表す CFG の各ノード情報となり，それを元にハッシュ値を計算し認証に用いる．また C-FLAT は静的にバイナリを解析した際のアドレスと，被認証プロセス実行時のアドレスが等しくなるように Address Space Layout Randomization(ASLR) を適用しない環境での運用を想定している．ハッシュ計算についての詳細は 3.3 で述べる．

3.2 バイナリ書き換え

C-FLAT では被認証プロセスのバイナリをプロセス実行前に動的に書き換え，制御フロー情報の取得とハッシュ計算プロセスへの送信を行う．書き換えには Capstone disassembly engine[8] というバイナリ書き換えツールを用い，プロセス実行前に読み込まれたバイナリに対して書き換えを行う．書き換えの際は，BB 遷移に関わる ARM アーキテクチャのリンク付き分岐命令である bl 命令をフックし，3.1 で述べた BB のアドレス情報を取得し，ハッシュ計算プロセスへの送信を行うように書き換える．

3.3 ハッシュ計算

本節では C-FLAT の基本となるハッシュ計算を述べた後，for 文や while 文などの繰り返し処理のハッシュ計算について述べる．

3.3.1 基本計算手法

ハッシュの計算は，BB 単位の CFG における各ノード単位で行う．あるノード n におけるハッシュ値 H_n 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 BB の先頭と末尾のアドレスである．また H_{prev} は前ノードのハッシュ値である．

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

ハッシュ計算の例を図 2 に示す．パス $[N_1 \rightarrow N_2 \rightarrow N_4]$ を通る場合，開始ノード N_1 のハッシュ値 H_1 は前ノードが存在しないため N_1 のノード情報から計算される．次にノード N_2 のハッシュ値 H_2 が前ノードのハッシュ値 H_1 と N_2 のノード情報 I_2 から計算される，またノード N_4 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 $[N_1 \rightarrow N_3 \rightarrow N_4]$ を通る場合も同様の手順で計算される．そのためノード N_4 でのハッシュ値 H_4 は各処理で通ったパスごとに異なる．これを用いて被認証プロセス実行時の制御フローを最終的なハッシュ値から特定する．

またこれらのハッシュ計算は ARM TrustZone の Secure World 上で行われる．保護領域内で計算を行うことで任意のプロセスからの改ざんを防止し，計算結果に正当性を持たせている．

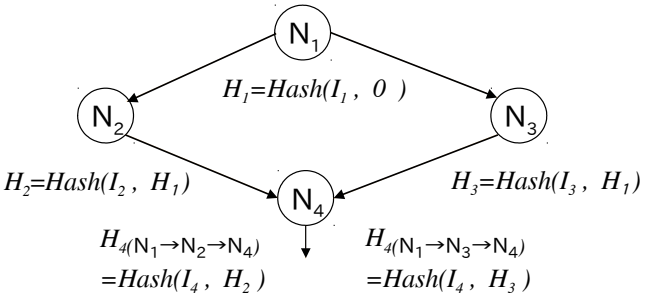

図 2 ハッシュ計算例

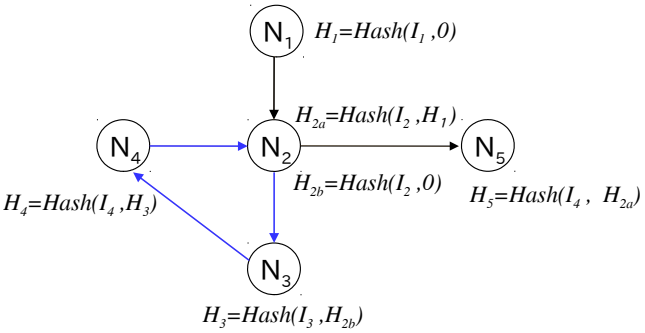

図 3 繰り返し処理を含んだハッシュ計算例

3.3.2 繰り返し処理

C-FLAT では for 文や while 文などのループを用いた繰り返し処理に対して計算量削減のための対応を行っている．C-FLAT のハッシュ計算手法は最終的なハッシュ値が実行された制御フローに対して一意に定まる．この計算手法を繰り返し処理を含むプログラムに適用した場合，繰り返しの回数ごとに最終的なハッシュ値が異なる値となる．このため正常な制御フローのハッシュ値を計算するコストが増大し，問題となる．

これを解決するために C-FLAT では繰り返し処理を行うループ内部と，それ以外の処理を分けてハッシュ値を計算する．計算例を図 3 に示す．図内の青色の矢印がループ処理を表す．ループの開始ノードである N_2 ではループ外部のハッシュ計算で用いる H_{2a} と，ループ内部で用いられる H_{2b} が計算される．H_{2a} は 3.3.1 で述べたハッシュ計算処理が行われる．対して H_{2b} は N_2 のノード情報のみから計算された，N_2 を始点としたハッシュ値である．この H_{2b} を元にループ内部のハッシュ計算を行うことで，ループ内部で独立したハッシュ値を計算し，ループ外のハッシュ値がループの影響を受けないようにしている．図 3 のプログラムの認証の際にはループ外部の最後的なハッシュ値 H_5 に加え，通ったループの終点のハッシュ値 H_4 と各ループの実行回数を用いループ内部の処理も認証する．

3.4 問題点

本節では C-FLAT を IoT に適用する際の問題点を述べ，次にその問題点の解決のアプローチの例として関連研究を述べる．

3.4.1 IoT に適用する際の問題点

C-FLAT を IoT デバイスに適用する際の問題点としてデバイス上でのハッシュ値の計算コストが大きいことが挙げられる．C-FLAT の文献 [4] によると，被認証プロセスのオーバーヘッドの約 70 %がハッシュ計算に由来するものであり，ハッシュ計算が被認証プロセスの実行速度に大きく影響を及ぼしている．また他の問題点として計算する制御フロー情報の多さが挙げられる．この問題はプログラム全体の制御フローが簡単な CFG で表すことができる場合は問題はない．しかし複雑な CFG を持つプログラムの場合は，RA の認証者が用いる被認証プロセスの正常な制御フロー情報の計算に膨大な計算コストがかかるため問題となる．また計算結果の正当性を保つため，デバイスでは ARM TrustZone による保護領域上でハッシュ計算行う．このためデバイス側に TEE 機構の導入が必要であり，デバイスのハードウェアに制限がかかるという問題もある．

3.4.2 LO-FAT

LO-FAT(Low-Overhead Control Flow ATtestation in Hardware)[9] はハードウェアベースで C-FLAT と同様の制御フローを用いた RA を行う手法である．C-FLAT では被認証プロセスバイナリに対して書き換えを行う必要があり，認証を行うプロセスに大きなオーバーヘッドが発生するという問題がある．LO-FAT では BB 単位の制御フロー情報取得とハッシュ計算処理をハードウェアを用いて実装を行い，前述した問題を解決している．

3.4.3 LB-FLAT

LB-FLAT(Log-Based Control Flow Attestation for Embedded Devices)[10] はログ情報として遷移元 BB の末尾アドレスと遷移先 BB の先頭アドレスのペアを記録し，異常な遷移が発生していないかを検証し RA を行う手法である．この手法は C-FLAT のハッシュ値の計算コストを削減するためにハッシュ値を用いずに認証を行う．ハッシュを用いないため C-FLAT と同様の手法で BB の遷移情報を取得した場合，デバイス内部に保存する認証データのログサイズが膨大になるという問題がある．この問題に対応するために LB-FLAT では記録するログ情報を，リンク付き分岐命令の階層情報を用いて削減している．この階層は呼び出す関数や分岐の深さを表すものであり，プログラム開始時点での階層を 0 とし，分岐命令が呼ばれる際にインクリメント，分岐先からのリターン命令が呼ばれる際にデクリメントされる．この階層の移動の際の自明なフローについて省略することでログデータの削減を行い，認証すべきフローを削減し計算コストの問題の解決を目指している．

4. 提案手法

本章では提案手法についての概要を述べ，その後提案システムの各部分での詳細を述べる．

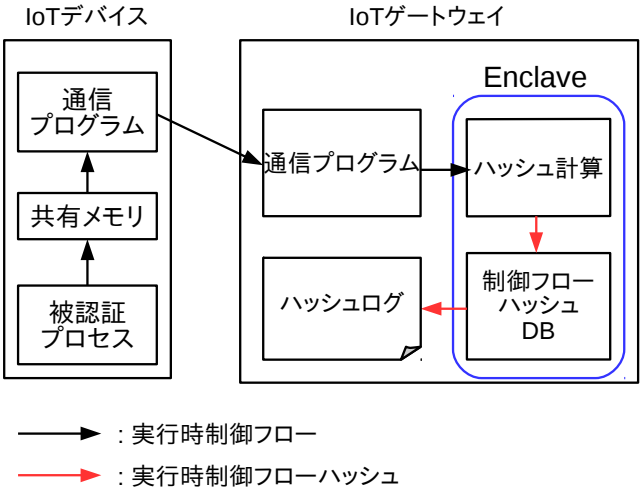

図 4 提案システム構成図

4.1 概要

本論文では，被認証プロセスの制御フローを用いた RA 手法を IoT ゲートウェイ (以下 IoTGW という) を用いて処理の分割を行い，IoT デバイスに適用する手法を提案する．3.4 節で述べたように C-FLAT のオーバーヘッドの大部分は制御フローを示すハッシュ値の計算に起因するものである．このため本論文では IoT デバイスを管理する IoTGW 上でハッシュ計算を行い，デバイス上のオーバーヘッドを削減し問題を解決する．IoTGW でのハッシュ計算は正当性の確保のため Intel SGX の保護領域である Enclave 上で行う．また C-FLAT や他の RA 手法ではデバイス上の認証データの計算結果に正当性を持たせるために TEE 技術を始めとした信頼点をデバイスに置くが，本提案では認証データの計算を IoTGW で行うためデバイス上に信頼点を配置しない．

提案システムの構成図を図 4 に示す．提案システムは IoT デバイスと IoTGW で構成される．各部分での処理の概要を述べる．IoT デバイスでは被認証プロセスから BB 単位の制御フロー情報を抽出し，共有メモリを介して IoTGW に対してデータ送信を行う通信プログラムへ情報を渡す．IoTGW では受け取った制御フロー情報をハッシュ計算プロセスに渡し，ハッシュ値を計算する．その後計算したハッシュ値を，正常な制御フローのハッシュ値を格納した制御フローハッシュデータベース (DB) で照合し，正常異常の判定を行い結果をログで出力する．この時用いる正常な制御フローは事前に被認証プロセスに対して解析を行い計算を行ったものである．ハッシュ計算と計算結果の照合は Intel SGX の Enclave 上で行い，計算結果と照合に用いる DB の改ざんを防止する．

4.2 IoT デバイス

IoT デバイスは被認証プロセスと通信プログラム，これらのプロセス間通信用の共有メモリで構成される．IoT デ

バイスでは 4.1 で述べたように，被認証プロセスの制御フロー情報の取得と，IoTGW への送信を行う．初めに被認証プロセスの制御フロー情報の取得について述べる．

被認証プロセスの制御フロー情報は，被認証プロセスのバイナリファイルに対して静的に書き換えを行い取得する．3.1 より C-FLAT では制御フロー情報取得の際，BB を識別するための情報として各 BB の先頭と末尾のアドレスを用い，それらを分岐命令をフックすることで動的に取得している．それに対し本提案では，各 BB の先頭にバイナリファイルでのアドレス値を埋め込み，埋め込まれたアドレスから各 BB を識別する手法を用いる．このため被認証プロセスのバイナリファイルに対して，BB 遷移の際に遷移先の BB を示すアドレスを共有メモリに出力するように書き換えを行う．被認証プロセス実行時に ASLR が有効な場合，出力されるアドレスは実際の実行アドレスとは異なるが，行われる処理は変化しないため，被認証プロセスの BB 遷移情報の取得には問題ないと考える．

次に通信プログラムでの動作について述べる．通信プログラムでは共有メモリから被認証プロセスの実行アドレスを読み込み，時系列順にまとめ IoTGW へ送信する．時系列順に実行アドレスをまとめることで，被認証プロセスの実行時の制御フローを表すデータとなる．送信は一定時間ごとに行い，送信間隔は被認証プロセスからの実行アドレス情報の取得量と間隔によって適宜変更する．

4.3 IoT ゲートウェイ

IoTGW は通信プログラムとハッシュ計算プロセス，被認証プロセスの正常な制御フローに基づくハッシュ値を格納した制御フロー DB で構成される．通信プログラムでは IoT デバイスから制御フロー情報を取得し，ハッシュ計算プロセスに受け渡す．次にハッシュ計算プロセスでは，受け取った制御フロー情報からハッシュ値を計算する．またハッシュ計算の際に必要となる前ノードのハッシュ値は Enclave 内のハッシュ計算プロセスで保存される．最後に計算されたハッシュ値と，予め計算した正常な制御フローのハッシュ値を照合し，正常もしくは異常の判定結果をログファイルに出力する．

5. 実装

本章では作成したシステムの実装を述べる．

5.1 被認証プロセス書き換え

本提案では被認証プロセスに対して，共有メモリの作成と実行アドレスの出力，ループ検知の機能を追加するよう書き換えを行った．本節ではこれらの書き換えの実装について述べる．初めに書き換えの概要について述べたあと，被認証プロセスバイナリに対する書き換えによって追加する動作の詳細について順に述べる．

5.1.1 書き換え概要

本提案ではバイナリ書き換えツールとして PEBIL(PMaC's Efficient Binary Instrumentation Toolkit for Linux)[11] を用いた．コード挿入前には被認証プロセスの BB 遷移を把握するため，PEBIL を用いて対象コードを解析する．その後解析した BB 遷移箇所に対して実行アドレスの出力処理とループの検知処理を追加する．また共有メモリ作成処理をプロセス開始時に行うように書き換える．

5.1.2 共有メモリ作成

共有メモリ作成処理は main 関数以下の全 BB 遷移を取得するために，main 関数実行前に行われるようにコード挿入を行う．共有メモリ作成は Boost library API[12] を用いて行い，共有メモリ内の構造は Boost liblary の Lock Free Queue 構造を用いた．Lock Free とは排他制御を行わずに，複数のプロセスが同時並列に共有メモリにアクセスすることを可能にするアルゴリズムのことである．排他制御を行う場合，片方のプロセスがアクセスしている間にもう一方のプロセスは待機状態になるため，被認証プロセスでのオーバーヘッドが増加する．そのため，Lock Free Queue を用いた．また今回の実装では Queue のサイズを超える書き込みが発生した際には，被認証プロセスが書き込み完了まで待機する．このため被認証プロセスの BB 遷移の時間当たりの回数によって Queue サイズを適宜変更する必要がある．

5.1.3 制御フロー情報の出力

被認証プロセスでの制御フロー情報の出力は，BB を示すアドレスを共有メモリに書き込む出力関数の追加と，各 BB の先頭に出力用関数を呼び出す処理を追加することで実装する．書き換えの例を図 5 と図 6 に示す．図 5 のコードは，中身が空の関数 f0 を呼び出した時の処理を抜粋したものであり，図 5 のコード部分で 1 つの BB となる．図 6 では黒枠の部分が書き換え前の処理に対応し，赤枠部分が書き換えで追加された処理となる．図 6 から元の処理の実行前に，BB を示すアドレスの出力処理が行われることが分かる．また本提案では制御フロー情報の出力処理は main 関数とユーザー定義関数内を対象とし，main 関数実行前の処理や，ライブラリ関数内の動作は書き換えの対象外とした．

5.1.4 ループ検知

3.3.2 より C-FLAT と同様のハッシュ計算を行うには，for 文や while 文などのループ処理を識別し計算を行う必要がある．このためハッシュ計算を行う IoTGW 上で，被認証プロセスから出力された制御フロー内のループ処理を識別する必要がある．本提案ではカウンターを用いてループ処理を識別する．ループの開始点にカウンターをインクリメント，ループ終了点にデクリメントする処理を書き換えによって追加し，カウンターの値を制御フロー情報と共に出力する．この出力されたカウンター情報の変化を元に

```
Function f0:
004004d6        PUSH        RBP
004004d7        MOV         RBP,RSP
004004da        NOP
004004db        POP         RBP
004004dc        RET
```

図 5　書き換え前コード

```
Function f0:
004004d6        JMP         00948044
   ⋮
00948044        PUSH        R15
   ⋮ //出力用関数呼び出し前処理
00948079        JMP         00948b30

0094807e        PUSH        RBP
0094807f        MOV         RBP,RSP
00948082        NOP
00948083        POP         RBP
00948084        RET

//出力用関数呼び出し
00948b30        CALL        00948741
00948b35        JMP         0094807e
```

図 6　書き換え後コード

IoTGW ではループの開始と終了を推定する．また書き換えを行うループ処理の開始点と終了点は PEBIL のループ検知ツールから取得する．

5.2　IoT デバイス・IoT ゲートウェイ間の通信

IoT デバイス上の通信プログラムでは送信する制御フロー情報を共有メモリから取得し IoTGW に送信するよう実装を行った．送信する情報は BB を示すアドレスの他に，ループの階層情報とデバイス識別用の ID がある．これらの情報を Google Protobuf[13] を用いてシリアライズし IoTGW へ送信する．

また IoT デバイスと IoTGW 間の通信は TCP/IP を用いて行う．本提案の実装では送信内容の暗号化や SSL を用いた通信は用いていない．しかし暗号通信を用いることで通信の際の改ざんを防ぎ，よりセキュアな認証が可能となる．そのため今後通信路の暗号化と，それに伴うオーバーヘッドを調査し，システムに適用可能か検討する．

5.3　ハッシュ計算

ハッシュ計算は Intel SGX によって作成した保護領域である Enclave 上で行う．2.2.2 より，Intel SGX の保護領域の呼び出しコストは TrustZone より小さい．そのため C-FLAT に比べ，本提案の保護領域呼び出しに由来するオーバーヘッドは小さくなる．また計算に用いるハッシュ関数には SHA256 を用い，ハッシュ計算は 3.3 で述べた方法で行う．本提案の実装は break 文と再帰的関数呼び出しに対してはループ処理の都合上未対応となっている．このため評価ではこれらを含まないプログラムを用いた．

表 1　評価環境

	IoT ゲートウェイ用 PC	IoT デバイス用 PC
CPU	Core i5-7500 @ 3.4GHz	Core i3-4130 @ 3.4GHz
RAM	8G	8G
OS	ubuntu16.04	ubuntu18.04
Intel SGX	○	×
SGX_SDK	2.7.1	-

5.4　制御フローハッシュ DB

本提案では正常な制御フローのハッシュ値を格納するデータベースとして SQLite[14] を用いた．SQLite は主にアプリケーションへ組み込み用途で用いられる軽量高速な DB である．本提案では被認証プロセスとして比較的小さなプログラムを想定しているため SQLite を用いた．また実装の際には SQLite を Intel SGX の Enclave 上に展開する手法 [15] を用い，セキュアに DB を展開するよう実装を行った．

5.5　正常データ作成

正常データの作成は被認証プロセスのバイナリを事前に解析して行った．解析には Dyninst[16] を用いて作成した CFG を用い，対象である main 関数内とユーザー関数内の CFG を抽出し，正常な制御フローのハッシュ計算を行った．計算したハッシュ値は SQLite のデータベースファイルに保存し，RA に用いる．

6.　評価

本提案のシステムを用いて評価を行った. 本章では評価項目に対する結果について述べる．評価環境を表 1 に示す．また評価項目を以下に挙げる．

- IoTGW で被認証プロセス実行時の制御フローを用いた異常検知の確認
- 被認証プロセスに対するシステム適用時のオーバーヘッド
- IoTGW におけるハッシュ値の計算時間

IoTGW での制御フローを用いた異常検知の確認はスタックバッファオーバーフローを用いて確認を行った．被認証プロセスに対する提案手法適用時のオーバーヘッドは適用前と適用後の被認証プログラムの実行時間を比較する．またハッシュ計算のオーバーヘッドは，被認証プロセスに対するシステム適用時のオーバーヘッド測定時の IoTGW におけるハッシュ計算に要する実行時間を算出する．

今回の評価では 2 台のコンピュータを用い，それぞれ IoTGW と IoT デバイスとして評価を行った．今回は被認証プロセスに使用するバイナリ書き換えツールが x86 アーキテクチャのみの対応であったため，IoT デバイス側の PC として Intel CPU 搭載の PC を用いた．実際の IoT デバイスと比較して高性能なものにはなるが，異常検知の確認とオーバーヘッドの確認には問題ないと考える．

```
1 void f1(char* input){
2     char buf[5];
3     memcpy(buf,input,strlen(input)*sizeof(char));
4 }
5 int main(int argc, char *argv[])
6 {
7     f1(argv[1]);
8     return 0;
9 }
```

図 7　評価用被認証プロセスコード

6.1　パスの正常，異常検知

評価に用いた被認証プロセスのコードを図 6.1 に示す．制御フローを元にした異常検知では制御フローを変更する手法としてスタックバッファオーバーフローを用いた．本提案の被認証プロセスの書き換えは main 関数とユーザー定義関数内の BB を対象として書き換えを行った．そのため書き換えを行った BB から BB への異常な遷移のみ検知可能である．評価では，gcc の stack protector オプションによって追加される，stack_chk_fail 関数へ移行させることで異常な遷移を発生させる．

評価時の被認証プロセスの関数 f1 内の処理の BB 遷移を図 8 に示す．図 8 では評価の際の実行フローを実線矢印で表し，正常フローを黒色の矢印とノードで示している．またスタック破壊の場合の異常フローを赤色の矢印とノードで示している．評価の際に IoTGW で計算，出力されたハッシュログから図 8 で示した BB 遷移に関わる部分を抜粋したものを図 9 に示す．ハッシュログは正常 DB のデータと一致しないデータに対してはログの先頭にラベル付が行われている．なお，ハッシュ値はスペースの関係で一部省略してある．図 9 より，異常な BB 遷移前の正常な BB 遷移時のハッシュ値は正常と判断され，異常な BB 遷移後のハッシュ値は正しく異常と判定されていることが分かる．

また今後，書き換えを行った BB から任意のアドレスへの遷移を伴う異常の検知を行う必要がある．そのため関数からのリターンの際にプロセス実行時のリターンアドレスを追加で取得する必要があると考えられる．

6.2　システム適用時のオーバーヘッド

オーバーヘッド測定のために被認証プロセスの BB 遷移回数に対する，システム適用時と非適用時の実行時間を比較する．評価にはループ内で空の関数を呼び出すプログラムを用い，システム適用前後のループの実行時間を測定する．BB の遷移回数はループ回数を変化させることで目的の回数になるように調節し評価を行った．評価結果を表 2 に示す．評価結果より BB 遷移一回あたりに約 1μs のオーバヘッドが発生することが分かる．

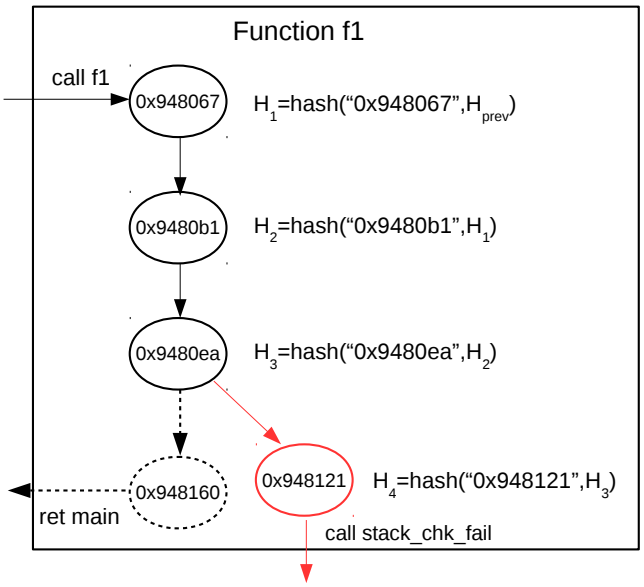

図 8　BB 遷移によるハッシュ計算例

```
H1 id=0 hash=3dd7f8503cc...
H2 id=0 hash=bff68fe10e2...
H3 id=0 hash=de1da1d82a6...
H4 [error hash]id=0 hash=1efe6315a97...
```

図 9　ハッシュログ例

表 2　実行時間

BB 遷移回数 (回)	10^2	10^3	10^4	10^5
適用前 (ms)	0.000802	0.00385	0.0352	0.354
適用後 (ms)	0.101	0.982	10.071	92.093
1 遷移あたりのオーバーヘッド (μs)	0.997	0.978	1.004	0.917

表 3　ハッシュ計算時間

BB 遷移回数 (回)	10^2	10^3	10^4	10^5
ハッシュ計算時間 (s)	0.0460	0.218	2.457	19.447
1 遷移あたりの計算時間 (ms)	0.460	0.218	0.246	0.194

6.3　ハッシュ値の計算時間

IoTGW におけるハッシュ計算に要した計算時間を測定した．6.2 のシステム適用時のオーバーヘッドの評価の際のハッシュ計算時間を元に評価を行う．評価結果を表 3 に示す．評価結果より 1BB 遷移あたりのハッシュ計算時間は約 0.2ms と判明した．これはデバイス上の被認証プロセスへのシステム適用時のオーバーヘッドの約 200 倍の時間に相当する．評価に用いた PC の性能差はあるが，ハッシュ計算を IoTGW 上で行うことで，制御フローベースの遠隔認証のハッシュ値計算のコストを削減できたことが確認できる．今回の評価では長時間連続で動作を行わないプログラムを用いたため問題なかったが，BB 遷移あたりのハッシュ計算時間が，BB 遷移あたりの被認証プロセスの実行時間より大きい場合，システムを連続で動作させ続けるとハッシュ値の計算が追いつかなくなるという問題が発生する．このため被認証プロセスの BB の遷移速度に対応した

性能を持つ IoTGW を用いる必要があることが判明した．

また今回の実装では 1BB 遷移ごとに Enclave にアクセスし，制御フロー情報の受け渡しハッシュ計算を行っている．そのため数十回分の BB 遷移を示す制御フロー情報をまとめて Enclave に受け渡し，Enclave へのアクセス回数を削減することで Enclave アクセスに関わるオーバーヘッドは削減できると考えられる．

7. 今後の課題

今後の課題として，まず一つ目により複雑な制御フローを持つプログラムへの提案システムの適用が挙げられる．今回の評価ではプログラムの全体のパスが小規模になるような単純なプログラムを用いて評価を行った．そのため，より複雑な制御フローを持つプログラムにシステムを適用した場合の正常な CFG の計算のコストや，プログラム全体のシステム適用前後の実行速度の変化を測定し，制御フローの複雑化に伴うシステム適用の際の影響を調査し対応を行う必要があると考えられる．

二つ目の課題として書き換えを行った BB から任意のアドレスへの遷移を伴う異常の検知が挙げられる．これは関数からのリターンの際にプロセス実行時のリターンアドレスを追加で取得し，取得したリターンアドレスと遷移すべき BB のアドレスを比較することで対応する．

また三つ目の課題として被認証プロセスと IoTGW における更なるオーバーヘッドの削減が挙げられる．オーバーヘッドの削減手法としては Enclave へのアクセス回数の削減する手法と，計算する制御フロー情報を削減しハッシュ計算の量を減らす手法が考えられる．Enclave へのアクセス回数削減は，1 アクセスごとに渡す制御フロー情報を増やすことで Enclave アクセスの回数を削減し対応を行う．計算する制御フロー情報の削減は LB-FLAT が行った RA の際に認証する制御フロー情報の削減が効果的であると考える．本提案手法ではライブラリ関数の call についても BB 遷移とみなし実装を行っている．このためライブラリ関数の実行の際には自明である制御フローが発生する．この自明な制御フロー情報の削減を行うことで書き換え箇所を減らし，オーバーヘッド削減に繋がると考えられる．

8. おわりに

本論文では，IoT デバイスに適した形での制御フローベースの RA システムを提案し，実装と評価を行った．本提案手法では制御フローを用いた RA 手法のオーバヘッドの大部分を占める，制御フローを示すハッシュ計算を IoT ゲートウェイ上で行うことで IoT デバイス上の被認証プロセスのオーバーヘッドを削減した．今後はより複雑なプログラムへのシステムの適用とその際に発生する問題点への対応を行う．また検知可能な異常の追加と，更なるオーバーヘッド削減の検討を行う．

謝辞

本研究の一部は，JSPS 科研費基盤研究 (C)19K11962 による助成を受けたものです．また，本研究の基本的なアイデアと実装に貢献していただいた Jerome PEYROUTAT-BASSE 氏に感謝いたします．

参考文献

[1] Gamblin, J.: Mirai-Source-Code, https://github.com/jgamblin/Mirai-Source-Code/blob/master/LICENSE.md (2007).
[2] Abera, T., Asokan, N., Davi, L., Koushanfar, F., Paverd, A., Sadeghi, A.-R. and Tsudik, G.: Invited - Things, Trouble, Trust: On Building Trust in IoT Systems, *Proceedings of the 53rd Annual Design Automation Conference*, New York, NY, USA, Association for Computing Machinery, (online), DOI: 10.1145/2897937.2905020 (2016).
[3] Prandini, M. and Ramilli, M.: Return-Oriented Programming, *IEEE Security Privacy*, Vol. 10, No. 6, pp. 84–87 (online), DOI: 10.1109/MSP.2012.152 (2012).
[4] Abera, T., Asokan, N., Davi, L., Ekberg, J.-E., Nyman, T., Paverd, A., Sadeghi, A.-R. and Tsudik, G.: C-FLAT: control-flow attestation for embedded systems software, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ACM, pp. 743–754 (2016).
[5] intel, C.: Intel SGX Homepage,https://software.intel.com/en-us/sgx.
[6] Alves T, D.: TrustZone: Integrated Hardware and Software Security (2004).
[7] Abadi, M., Budiu, M., Erlingsson, U. and Ligatti, J.: Control-Flow Integrity Principles, Implementations, and Applications, *ACM Trans. Inf. Syst. Secur.*, Vol. 13, No. 1 (online), DOI: 10.1145/1609956.1609960 (2009).
[8] Capstone: The Ultimate Disassembly Framework – Capstone – The Ultimate Disassembler.
[9] Dessouky, G., Zeitouni, S., Nyman, T., Paverd, A., Davi, L., Koeberl, P., Asokan, N. and Sadeghi, A.: LO-FAT: Low-Overhead Control Flow ATtestation in Hardware, *CoRR*, Vol. abs/1706.03754 (online), available from ⟨http://arxiv.org/abs/1706.03754⟩ (2017).
[10] Liu, J., Yu, Q., Liu, W., Zhao, S., Feng, D. and Luo, W.: Log-Based Control Flow Attestation for Embedded Devices, *International Symposium on Cyberspace Safety and Security*, Springer, pp. 117–132 (2019).
[11] Laurenzano, M. A., Tikir, M. M., Carrington, L. and Snavely, A.: Pebil: Efficient static binary instrumentation for linux, *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS 2010)*, IEEE, pp. 175–183 (2010).
[12] Beman Dawes, David Abrahams,Rene Rivera: boost C++ libraries, https://www.boost.org/.
[13] Google LLC: Google Protocol Buffers, https://developers.google.com/protocol-buffers/.
[14] SQLite: SQLite Homepage, https://www.sqlite.org/index.html.
[15] Xu, P.: SQLite database inside a secure Intel SGX enclave (2019).
[16] Project, P. T.: Dyninst Putting the Performance inHigh Performance Computing, https://www.dyninst.org/.