Vol.2020-OS-149 No.6
2020/5/28

関数ポインタの使用有無に着目した Intel CET による攻撃ガジェット削減

伊東 拓海[1]　石黒 健太[1]　河野 健二[1]

概要：C/C++ で記述されたアプリケーションの制御フローをのっとる攻撃が増えている．攻撃者はメモリ上の関数ポインタの値を不正に書き換えることで，ガジェット (gadget) と呼ばれるアプリケーション内のコード片をつなぎ合わせた攻撃コードを実行する．ガジェットとして利用できるコード片を関数単位に限定するハードウェア支援の 1 つに Intel CET (Control-flow Enforcement Technology) がある．Intel CET を利用しても全ての関数をガジェットとして利用できるため，チューリング完全であることが指摘されている．

本論文では関数ポインタを介して呼び出されることのない関数をガジェットとして利用できなくする手法を示す．シンボル情報をストリップしたバイナリを解析し，関数ポインタ経由で呼び出されることのない関数や，オブジェクト生成の行われていないクラスの仮想関数などをガジェットとして利用できないようにする．C/C++ で書かれたライブラリに対して本方式を適用したところ，54% 程度利用可能なガジェットを減らすことができた．

キーワード：ソフトウェアセキュリティ，CFI，バイナリ解析, Intel CET

1. はじめに

近年，C/C++ で記述されたアプリケーションの制御フローをのっとる攻撃が増えている．その脆弱性の例として，最近では WebkitGTK (CVE-2020-11793) [1] というソフトウェアや，Android (CVE-2020-0022) [2] がある．[1] では，use-after-free の脆弱性を用いることによって，攻撃者が任意のことを実行できる．[2] では，割り当てたメモリの範囲外に書き込むことができる脆弱性によって，攻撃者が権限昇格なしに Bluetooth を通じてリモート実行することができるようになる．制御フローをのっとる攻撃では攻撃者は，プログラム中のガジェット (gadget) と呼ばれるコード片を数珠繋ぎにすることで攻撃を行う．制御フローをのっとるための 1 つの手法として，関数ポインタを書き換えて間接ジャンプの飛び先を変える攻撃がある．間接ジャンプの飛び先を変えて攻撃を行うことを，forward-edge による攻撃という．forward-edge による攻撃の例として，jmp 命令を用いた Jump Oriented Programming (JOP) 攻撃 [3] や C++ における仮想関数呼び出しを悪用した Counterfeit Object Oriented Programming (COOP) 攻撃 [4] がある．逆に，関数の戻り番地を不正に書き換えることによって，ret 命令を利用して攻撃を行うことを backward-edge による攻撃という．backward-edge による攻撃の例として，Return Oriented Programming (ROP) 攻撃 [5], [6] がある．本論文では，backward-edge による攻撃はシャドウスタック (shadow stack) によって保護されているものとし，forward-edge による攻撃を防ぐことを考える．

制御フローをのっとる攻撃を防ぐための機構として，Control-flow Integrity (CFI) [7] がある．forward-edge による攻撃を防ぐため, CFI の 1 つとして, Intel 社は Control-flow Enforcement Technology (CET) [8] と呼ばれる CPU の命令拡張を提供している．CET を用いることによって，間接ジャンプの飛び先を制限することができる．CET を用いると，任意のコード片ではなく，ガジェットを関数単位に制限することができる．そのため，利用できるガジェットを大幅に制限することができる．しかし，関数をつなげて実行することで，チューリング完全になることが知られている [4], [9].

本論文では，Intel CET を用いた攻撃ガジェットの削減を提案する．ソースコードが入手可能であるとは限らないため，バイナリを解析対象とし，多くのバイナリがストリップされた形で提供されていることからストリップされたバイナリを対象とする．攻撃ガジェットを削減するための手法として，具体的には次の 2 点を達成する．

[1] 慶應義塾大学
Keio University

- 関数ポインタを介して呼び出されることのない関数をガジェットとして利用できなくする．
- C++ において呼び出されることのない仮想関数をガジェットとして利用できなくする．

ガジェットとして利用できる仮想関数を削減するために現段階では，インスタンス化されていないクラスの仮想関数をガジェットとして利用できないようにしている．

バイナリ解析フレームワークである ROSE [10] に提案手法の実装を行った．関数ポインタを介して呼び出されることのない関数をガジェットとして利用できなくするために，制御フローをたどり，プログラム中で使用されている関数ポインタを探索する．また，仮想関数を削減するために，制御フローをたどりオブジェクトが生成されている場所を探索する．そして，Intel CET を用いることによってオブジェクトが生成されていないクラスの仮想関数と関数ポインタを介して呼び出されることのない関数をガジェットとして利用できなくする．

本論文では，libstdc++ ライブラリ，libc ライブラリ，leveldb [11] ライブラリに対して実験を行った．結果，54 % 程度利用可能なガジェットを削減することができる．

本論文の構成は次のとおりである．第 2 章では，本論文の脅威モデルについて説明する．第 3 章では，本論文で提案する Intel CET による攻撃ガジェット削減の手法について説明する．第 4 章では，どのくらい本論文の提案手法によって攻撃ガジェットが削減できたかを説明する．第 5 章では，関連研究について説明する．第 6 章では，本論文のまとめを述べる．

2. 脅威モデル

ここでは，脅威モデルとして次のような前提を置く．まず，攻撃者がコード領域のコードを修正したり，新しいコードを挿入できないようにするために，保護されるアプリケーションには W ⊕ X が用いられているとする．つまり，攻撃者はコード領域は書き換え不可であり，また，プログラム内のメモリ管理に関するバグを利用して，ヒープ領域とデータ領域の任意のアドレスを任意の値に書き換えることが可能である．本論文では，保護対象のプログラムのバイナリのみがあるとする．また，保護対象のバイナリはシンボル情報がストリップされているものとする．これは，ソースコードが入手可能であるとは限らず，多くのバイナリがストリップされた形で提供されているためである．また，保護対象のバイナリに悪意のある攻撃コードは含まれないものとする．本論文では，indirect call/jmp 命令を用いて攻撃を行う forward-edge による攻撃を対象とする．また，関数の戻り番地を書き換える backward-edge を用いる攻撃に関しては out of scope とする．backward edge に関しては，シャドウスタックによって防ぐことができ，また，シャドウスタックは本論文の手法と異なり制御フローをたどらずに防ぐことができる．また，制御フローに影響を与えず．プログラム中の変数を修正したりリークさせたりする Data-Only 攻撃は out of scope とする．また，プログラムの実行中にコードが生成される Jit コンパイラも out of scope とする．

3. Intel CET による攻撃ガジェット削減

3.1 Intel Control-flow Enforcement Technology

制御フローをのっとる攻撃を防ぐ手法として，Control-flow Integrity (CFI) [7] がある．CFI によって間接ジャンプ時の飛び先を制限したり，return 命令時に不正なところに飛ばないようにすることができる．forward-edge による攻撃を防ぐための CFI の 1 つとして，Intel 社は Control-flow Enforcement Technology [8] と呼ばれる CPU の命令拡張を提供している．Intel CET は，indirect call/jmp 時の飛び先を制限する機構と，関数の戻り番地を正しくすることができるシャドウスタック (shadow stack) を持つ．

indirect call/jmp 時の飛び先を制限することができる forward-edge CFI から説明する．これは，indirect call/jmp 時には，`endbr` 命令があるところにしか飛ぶことができないというものである．`endbr` 命令とは，forward-edge CFI を実現するために，Intel 社が新しく追加した命令である．indirect call/jmp 時に `endbr` 命令でないところに飛んだ場合，ハードウェアによる例外が発生し，アプリケーションプログラムが終了する．通常 CET がない場合，攻撃者が攻撃時に使用するガジェットの始めのアドレスとして，任意のアドレスをとることができる．つまり，攻撃者は間接ジャンプ時に関数の途中や命令の途中と言った任意のアドレスに飛ぶことができる．しかし，CET を用いると，`endbr` 命令があるところにしか indirect call/jmp 時に飛べなくなる．例えば，関数のエントリポイントに `endbr` 命令を置くことによって，関数の途中や命令の途中に indirect call/jmp 時に飛べなくすることができる．そのため，CET を用いることによって利用できるガジェットを大幅に制限することができる．しかし，関数をつなげて実行することによって，チューリング完全になることが知られている [4], [9]．また，現状 gcc [12] や llvm [13] といったコンパイラでは全ての関数のはじめに `endbr` 命令が入る．そのため現状では，関数単位のガジェットをつなげて行う攻撃 [4] を CET で防ぐことはできない．

Intel CET には shadow stack も持つ．shadow stack を用いることによって，関数の戻り番地を書き換えて攻撃を行う backward-edge による攻撃を防ぐことができる．本論文では，backward-edge は対象としていない．しかし，backward-edge は CET の shadow stack で防ぐことができる．

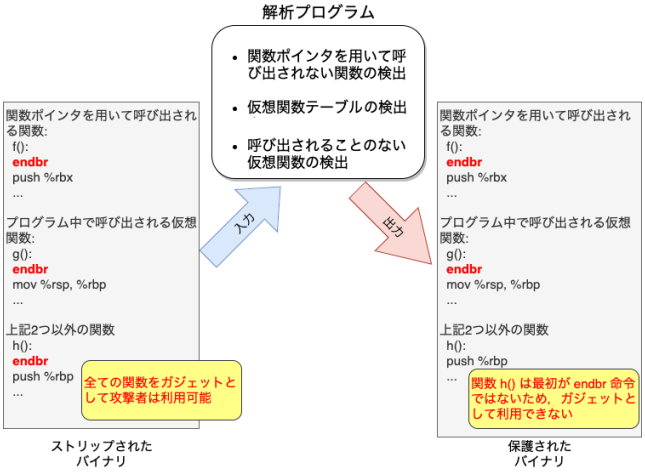

図 1 本論文における解析システムの概略図

3.2 概要

この節では，本論文で提案するシステムの概要について説明する．第 3.1 節で説明したように，現状では CET を用いても，すべての関数をガジェットとして使用できてしまう．そのため，本論文では利用可能な攻撃ガジェットをさらに削減するために，以下の 2 点を達成するようにする．

- 関数ポインタを介して呼び出されることのない関数をガジェットとして利用できなくする．
- C++ において呼び出されることのない仮想関数をガジェットとして利用できなくする．

本論文で提案する解析システムの全体像を図 1 に示す．解析プログラムの入力として，ストリップされたバイナリを入力として受けとる．解析プログラムでは，関数ポインタを用いて呼び出されない関数の検出と呼び出されることのない仮想関数の検出を行う．また，本論文ではシンボル情報がストリップされたバイナリを用いるので，仮想関数テーブルの検出を行う．解析プログラムの出力として，関数ポインタを介して呼び出されない関数をガジェットとして使用できなくしたバイナリが出力される．図 1 のように，関数ポインタを用いて呼び出される関数の 1 つとして `f()`, プログラム中で呼び出される仮想関数の 1 つとして `g()`, そのどちらでもない関数の 1 つとして `h()` があるとする．関数ポインタを用いて呼び出される関数とプログラム中で呼び出される仮想関数には，`endbr` 命令が入る．それ以外の関数は，関数ポインタを介して呼び出されることがない関数なので，`endbr` 命令が入らない．これによって，プログラム中で使用されない関数および間接ジャンプによって呼び出されない関数を攻撃者にガジェットとして使用できなくすることができる．

3.3 関数ポインタを用いて呼び出す関数の探索

関数ポインタを用いて呼び出される関数を特定する方法の 1 つとして，間接ジャンプ命令を探し，そこを起点にデータフロー解析をすることがあげられる．しかし，この解析方法は，制御フローを長い距離さかのぼることになることもあり，経路爆発問題が起こるため難しい．そのため，データフロー解析を避けつつ，間接ジャンプで呼び出されうる関数を保守的に見つけてくる必要がある．この節では，その方法について説明する．

```
1 /* 0x456890 は関数 f() のアドレス */
2 mov $0x456890, %rax
3 ...
4
5 0x456890: f()
6 ...
```

コード 1 `mov` 命令を用いて関数ポインタを参照する例

```
1 /*(%rip+0x4560) は関数 g() が始まるアドレス */
2 /* %rip は lea 命令の次の命令のアドレス
      (0x450000)が入っている*/
3 0x44fff9: lea 0x4560(%rip), %rax
4 0x450000: ...
5 ...
6
7 0x454560: g()
8 ...
```

コード 2 `lea` 命令を用いて関数ポインタを参照する例

まず，関数ポインタが格納される場所として，コード領域とデータ領域の 2 種類がある．

コード領域に格納される場合は，コード 1 の 2 行目やコード 2 の 3 行目のように即値やプログラムカウンタに対する相対アドレッシングとして，コード中に埋め込まれる．データ領域に格納される場合は，リンカなどによってあらかじめデータ領域に格納される．また，本論文で対象とするバイナリは，gcc や llvm などのコンパイラによって生成されたバイナリであって，関数ポインタを計算によって求めることはしていないと仮定する．

データ領域に格納されている関数ポインタの探索方法について説明する．データ領域に格納されている関数ポインタを探索するときに，それが本当に関数ポインタなのか他のデータであるのかを判断するのは難しい．これを判断するためには，データ領域から読み込まれた値が間接ジャンプ命令のオペランドとして使われるかどうかデータフロー解析をしなければならない．上述した通り，データフロー解析をすることは容易ではない．そのため，本論文ではデータ領域中のコード領域に相当する値は全て保守的に関数ポインタであるとみなし，全て間接呼び出しで呼び出しうるとみなす．ストリップされたバイナリでも，コード領域，データ領域といった領域のアドレスの範囲を特定することはできる．そのため，この手法をとることは可能で

ある．

コード領域に格納されている関数ポインタの探索方法について説明する．コード領域に格納される場合は，コード 1 の 2 行目やコード 2 の 3 行目のように即値やプログラムカウンタに対する相対アドレッシングとして，コード中に埋め込まれる．データ領域から関数ポインタを読みこんで間接ジャンプ命令でその関数を呼び出す場合もある．しかし上述したように，データ領域中の関数ポインタとなりうる値は全て間接呼び出しされるものとみなしているので，ここでは考える必要がない．コード領域に関数ポインタが格納される場合，コード 1 のように `mov` 命令を用いて関数ポインタを即値で表現したり，コード 2 のように `lea` 命令を用いてプログラムカウンタに対する相対アドレッシングで表される．このような命令を探し，そこを起点にデータフロー解析を行うことによって間接ジャンプ命令によってその関数が呼び出されるかどうかを判断する方法があるが，データフロー解析をすることは容易ではない．そのため本論文では，コード 1 とコード 2 のような命令で見つけた即値やプログラムカウンタに対する相対アドレッシングがコード領域に相当する値だった場合，間接呼び出しで呼び出されるとする．

コード領域に格納されている関数ポインタを探索するために，制御フローをたどる．バイナリのエントリポイントとデータ領域に格納されている関数ポインタを起点に制御フローをたどる．関数ごとに解析をし，コード 1 とコード 2 のような命令を探す．コード 1 のような命令を発見した場合は即値を，コード 2 のような命令を発見した場合はプログラムカウンタに対する相対アドレッシングの値を解析すべき関数リストに追加する．また，`call` 命令を見つけ，それが direct call であった場合，そのターゲットアドレスを解析すべき関数リストに追加する．

プログラムの実行中に動的に生成されたデータ構造に関数ポインタを格納する場合がある．ただし，その場合もコード 1 やコード 2 のどちらかの方法によって取得してから格納している．そのため，上記で述べたような手法をとることに問題はない．

本論文では，ストリップされたバイナリを対象としている．そのため，バイナリ内の全ての関数の位置を知ることは容易ではない．そのため，コード領域に相当する値をデータ領域・コード領域から発見した場合は，その値を関数のアドレスと保守的にみなして解析をしている．

この提案手法によって，確実に関数ポインタを用いて呼び出されない関数を特定することができる．そのような関数は，ガジェットとして利用できなくするために `endbr` 命令を入れないようにする．

3.4 オブジェクトが生成されているクラスの探索

第 3.3 節で示した手法ではデータ領域内の関数ポインタは保守的に全て使われているとみなしている．この節では，C++ のセマンティクスに着目して，データ領域内の関数ポインタのうち間接呼び出しで使われていないものをガジェットとして利用できなくする方法を示す．仮想関数は仮想関数テーブルと呼ばれるデータ領域の中にある関数ポインタの配列を用いて呼び出される．具体的には，「オブジェクトが生成されてから仮想関数が呼び出される」という点に着目して，プログラムの中でインスタンス化されていないクラスの仮想関数をガジェットとして利用できないようにする．

オブジェクトが生成されたかどうかを確認するためには，仮想関数テーブルのアドレスが重要になる．そのため，仮想関数テーブルの検出について説明をし，オブジェクトの生成されているクラスの探索についての説明する．

3.4.1 仮想関数テーブルの検出

本論文ではストリップされたバイナリを対象としているため，シンボル情報から仮想関数テーブルを抽出することができない．そのため，仮想関数テーブルのバイナリ中の位置を探す必要がある．

仮想関数テーブルの一般的な構造は，図 2 のようになる．ヘッダとして，Offset-to-top フィールドと Run-Time Type Information (RTTI) フィールドがある．その下に仮想関数のアドレスの配列が続く形になる．Offset-to-top フィールドは多重継承している時に 0 でない値をとり，そうでない時は 0 になる．RTTI フィールドの値は実行時型情報があるときは data セクションを，実行時型情報がない場合は 0 を指している．

関数テーブルを探す手法として，MARX [14] の手法をもとにした．仮想関数テーブルを探す方法として，以下の 6 つを条件とした．

(1) 仮想関数が read-only セクション (`.rodata`, `.data.rel.ro`, `.rdata` セクションなど) にあるか．
(2) 仮想関数テーブルの中の関数配列の始めが参照されているか
(3) Offset-to-Top フィールドの値が適切な値か
(4) RTTI フィールドの値が data 領域を指しているか 0 か
(5) 関数のエントリポイントがコード領域を指しているか
(6) 仮想関数テーブルの中の関数配列の最初の 2 つのエントリは 0 になることがある．

(3) については，-0xffffff から 0xffffff の間にあれば適切な値とした．この条件は MARX でも用いていられており，これは実験的に Prakash [15] が導いた値である．(3), (4) の条件を満たし，RTTI フィールドの直後 (条件 (6) を考慮した場合 2 つまで 0 をはさんでも問題ない) がコード領域を指していたら，そこから仮想関数の配列が始まるとする．仮想関数の配列の始めから順番に値を読み取っていき，コード領域を指さない値が見つかった時に，直前の

図 2 一般的な仮想関数テーブルの構造

```
1 /* 0x756890 は仮想関数テーブルの先頭アドレス*/
2 mov $0x756890, %rax
3 /* ヘッダの分ずらす */
4 add $0x10, %rax
5 /* オブジェクトの中の仮想関数テーブルを指すポインタの初
      期化をする*/
6 mov %rax, (%rbx)
```

コード 3 mov 命令を用いてオブジェクトを生成する例

```
1 /*(0x32290+%rip) は仮想関数テーブルの中の関数配列の
      先頭のアドレス */
2 lea 0x322900(%rip), %rax
3 /* オブジェクトの中の仮想関数テーブルを指すポインタの初
      期化をする */
4 mov %rax, (%rbx)
5 ...
```

コード 4 lea 命令を用いてオブジェクトを生成する例

コード領域を指している場所を仮想関数の終端とする．(6) だが，これは抽象クラスの仮想関数テーブルの場合，仮想関数テーブルの中の関数配列の最初の 2 つのエントリが 0 になることがある．そのため，(6) の条件を入れている．

3.4.2 オブジェクトが生成されているクラスの探索

この節では，オブジェクトが生成されていないクラスの探索方法について説明する．C++ において，オブジェクトが生成される時，オブジェクトの中にある仮想関数テーブルを指すポインタが初期化される．このオブジェクトを生成するときの命令パターンには一定のパターンがある．そのため，コード 3，コード 4 のような仮想関数テーブルを初期化する命令群を探す．コード 3 のように mov 命令を用いている場合は，仮想関数テーブルの最初のフィールドのアドレスをレジスタに代入し，add 命令によってヘッダの分をずらし，関数のアドレスの配列の先頭をオブジェクトに代入する．

多重継承しているクラスの場合，仮想関数テーブルを 2 個以上持つことがある．多重継承しているクラスの場合は，コード 3 の 1 行目，コード 4 の 1 行目のようにアドレスをレジスタに代入した後に，add 命令でオフセットを調整して 2 つ目以降の仮想関数テーブルのアドレスをレジスタに代入する場合がある．こういったケースにも注意してオブジェクト生成されているクラスを探索する．

コード領域の制御フローの辿り方は基本的に 第 3.3 節で説明した時と同じ辿り方である．ただし，オブジェクトが生成されたと確認された場合は，そのクラスの仮想関数を全て解析すべき関数リストに追加する．また，第 3.3 節では，データ領域にある関数ポインタは全て使われているものとして解析を始めているが，ここでは，データ領域にある仮想関数以外の関数ポインタが使われていると仮定して解析を始める．また，データ領域のオブジェクト上にコンパイラによってオブジェクトの仮想関数テーブルを指すポインタが初期化されている場合がある．そのため，データ領域を読み取り，仮想関数テーブルの値を読み取った場合，そのクラスのオブジェクトは生成されているとして解析する．

3.5 実装

バイナリ解析ツールである ROSE [10] を用いて実装を行った．ROSE のバージョンとして，0.9.10.200 を用いた．

図 1 にもあるように，関数ポインタを介して呼び出される可能性がある関数には endbr 命令が入る．現在，gcc などのコンパイラでコンパイルすると，全ての関数の始めに endbr 命令が入る．そのため，今回実装した際には，関数ポインタを介して呼び出されることのない関数の endbr 命令を nop 命令 4 つ分に書き換えた．これは，endbr 命令が 4 バイト命令だからである．Intel 社は endbr 命令について，「プログラムの状態に影響を与えない，レジスタに対して読み書きをしない」[8] としており，endbr 命令を nop 命令で書き換えることについては問題ない．

共有ライブラリの場合，アプリケーションによって使う関数，使わない関数が異なる．本論文では，その複雑さを回避するため，ライブラリが静的リンクされたバイナリを対象としている．ただし，ライブラリの動的リンク時にコードの書き換えを行ってからリンクすれば動的リンクにも対応可能である．

4. 実験

実験では，本提案手法を用いることによってどの程度ガジェットが減るのかを調べる．具体的には，本提案手法を用いる前と後で，バイナリの中の endbr 命令の数がどの程度減ったかを調べる．ライブラリの対象として，libc ライブラリ，libstdc++ ライブラリ，glibc ライブラリ，leveldb ライブラリ [11] を対象として実験を行った．実験環境とし

表 1 解析環境

OS	Ubuntu 18.04.2 LTS
CPU	Intel(R) Xeon(R) CPU E5-2430 v2 @ 2.50GHz * 6 Cores
Host Memory	16 GB
コンパイル時の gcc のバージョン	8.3.0
コンパイル時の最適化オプション	-O2
glibc のバージョン	2.28
leveldb のバージョン	1.20

表 2 glibc, libstdc++, libc ライブラリを用いたときの結果

状態	`endbr` 命令の数 (削減率)
元の `endbr` 命令の数	5417
第 3.3 節の手法を用いた場合	2630 (-51.4%)
第 3.4 節の手法を用いた場合	5289 (-2.4%)
上記 2 つの提案手法を同時に用いた場合	2502 (-53.8%)

表 3 leveldb ライブラリを追加で用いたときの結果

状態	`endbr` 命令の数 (削減率)
元の `endbr` 命令の数	6545
第 3.3 節の手法を用いた場合	3322 (-49.2%)
第 3.4 節の手法を用いた場合	6416 (-2.0%)
上記 2 つの提案手法を同時に用いた場合	3193 (-51.2%)

て，表 1 の環境で実験を行った．

まず，始めに glibc, libstdc++, libc ライブラリを静的リンクした時の結果を表 2 に示す．ユーザ側のアプリケーションプログラムとして，クラスのオブジェクトを生成し，入出力を用いているプログラムを用いた．結果として，関数ポインタを用いて呼び出されない関数を探す手法 (第 3.3 節) で 50% 以上ガジェットを削減できることが判明した．direct call のみでしか呼ばれない関数もあるため非常に多くガジェットを削減できる．また第 3.3 節に加えて，C++ で生成されていないオブジェクトの仮想関数も呼び出せないようにした (第 3.4 節の手法) 結果，53.8% のガジェットを削減することができた．表 2 を見て分かる通り，本論文で示した 2 つの提案手法のうち 第 3.3 節で提案した関数ポインタを用いて呼び出されない関数をガジェットとして呼び出せなくする手法がガジェット削減に対して効果が大きいということができる．

またもう 1 つの実験として，上記の 3 つのライブラリに加えて leveldb ライブラリを静的リンクした．そのときの結果を表 3 に示す．ユーザ側のアプリケーションプログラムとして，データベースの登録・削除・変更・閲覧をするプログラムを用いた．結果，leveldb ライブラリをリンクしていない時の実験結果と同様に，関数ポインタを用いて呼び出されない関数を探す手法 (第 3.3 節) で 50% 近いガジェットを削減できることが判明した．また，オブジェクトを生成しないクラスの仮想関数をガジェットにしない手法の方だが，これは leveldb ライブラリ以外の仮想関数がガジェットとして使えなくなっており，leveldb の仮想関数は全てガジェットとして利用できた．これは，leveldb ライブラリに関するクラスは 24 クラスあるのだが，そのうち 20 クラスはデータベースの初期化の処理でオブジェクトが生成されていたことが理由の 1 つとして考えられる．また，leveldb ライブラリをリンクしていない時の実験結果と同様に，本論文で示した 2 つの提案手法のうち 第 3.3 節で提案した関数ポインタを用いて呼び出されない関数をガジェットとして呼び出せなくする手法がガジェット削減に対しての効果が大きいということができる．

5. 関連研究

制御フローをのっとる攻撃を防ぐために，様々な CFI[7] の論文が出されている．本章では CFI に関する論文について紹介する．

本論文では，関数ポインタを介して呼び出されない関数をガジェットとして使用できなくした．同じように実行中に使わることのない関数をガジェットとして使用できなくする論文として，Nibbler [16] がある．本論文では，CET [8] というハードウェア拡張を用いて攻撃ガジェットの削減をしているが，Nibbler ではハードウェアを用いずにソフトウェアだけで攻撃ガジェットの削減に取り掛かっている．具体的には，共有ライブラリ中で使われないと分かった関数の中の中の命令をトラップ命令で全て書き換えることによって，その関数をガジェットとして使用できなくしている．ただし，Nibbler ではストリップされていないバイナリを対象として解析をしている．しかし，現在の世の中のライブラリはストリップされたバイナリであることが多い．これは，世の中のプログラムのバイナリに対して Nibbler を適用することが難しくなることを意味している．

また，CFI の種類として，ソースコードを解析することによって CFI を実現する手法とソースコードを用いずにバイナリ解析によって CFI を実現する手法の 2 種類がある．ソースコードを解析することによって実現する CFI の例として，μCFI [17] と VTrust [18] がある．μCFI では，ソースコードを解析することによって，indirect call/jmp 時による飛び先の数を限定している．特に μCFI では，制御フローに関係する重要な変数を特定し，その変数の値をプログラム実行中に読み取ることによって，indirect call/jmp 時のターゲット先を 1 つに絞っている．VTrust では，仮想関数呼び出し時に，呼び出せる仮想関数を継承関係があるクラスのみに制限し，さらに引数の数が一致している関数に制限している．ソースコードを用いて解析する利点として，引数の数や C++ におけるクラスがはっきりとわかるというのがある．しかし，短所としてソースコードが入手できないアプリケーションやライブラリの場合 CFI をすることができなくなる．また，ソースコードを用いずにバイナリを解析することによって実現する CFI の例として，TypeArmor [19] と VPS [20] がある．TypeArmor では，呼び出し元でセットした引数の個数と呼び出された関数で使われた引数の数を比較することによって forward-edge における CFI を実現している．VPS は，C++ の仮想関

数呼び出し時に違うクラスの仮想関数を呼び出せなくしている．VPS では，C++ におけるオブジェクト生成時に仮想関数テーブルを指すポインタをオブジェクトだけでなく，shadow memory にもコピーする．そして，仮想関数呼び出し時にオブジェクトの中の仮想関数テーブルを指すポインタと shadow memory にある値を比較して同じかどうかを確認する方法で CFI を実現している．

同じ CET を用いて CFI を実現している論文として，Hurdle [21] がある．Hurdle では，Intel CET によってガジェットが関数単位になっていることを前提に，それだけでは攻撃ができてしまうということで，さらにガジェットを削減している．また，Hurdle では関数の戻り番地のアドレスを記録するために，CET の shadow stack を用いている．本論文では，アプリケーション全体で関数ポインタを介して呼び出されていない関数に対して indirect call/jmp でとべなくしている．それに対し，Hurdle では関数の呼び出し元ごとに呼び出せる関数を絞っている．具体的には indirect call/jmp 命令に至るまでのヒストリを利用することによって，呼び出せる関数を絞っている．

この章であげた Hurdle 以外の CFI の論文は全てハードウェア拡張を用いずに，ソフトウェアのみで実装したものである．そのため，実験するアプリケーション・ライブラリによってはオーバーヘッドが大きくなる場合がある．(最悪の場合の実行時間のオーバーヘッド µCFI: 47.6%, VTrust: 8.0%, TypeArmor: 22.2%, VPS: 35%) 本論文では，ハードウェア拡張を用いてガジェットを削減している．さらに，本論文では実行時に解析を行うことはなく，解析を行うのは実行前に行う静的解析のみである．まだ Intel CET の実機が出ていないため，本論文では実行時間のオーバーヘッドを測定しなかったが，上述したような論文と異なり，本提案手法ではどのアプリケーション・ライブラリに対しても実行時間のオーバーヘッドを限りなく低くできることが期待できる．

6. まとめ

C/C++ で記述されたアプリケーションの制御フローをのっとる攻撃が増えている．攻撃者はメモリ上の関数ポインタの値を不正に書き換えることで，ガジェット (gadget) と呼ばれるアプリケーション内のコード片をつなぎ合わせた攻撃コードを実行する．

本論文では，Intel CET を用いて攻撃ガジェットを削減する手法を提案した．具体的には，関数ポインタを介して呼び出されることのない関数と，C++ において呼び出されることのない仮想関数をガジェットとして利用できなくした．

本論文の提案手法を C/C++ で書かれたライブラリのシンボル情報がストリップされたバイナリに対して適用したところ，54% 程度利用可能なガジェットを削減することができた．

謝辞 本研究は，JST, CREST, JPMJCR19F3 の支援を受けたものである．

参考文献

[1] : CVE - CVE-2020-11793, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-11793.

[2] : CVE - CVE-2020-0022, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-0022.

[3] Bletsch, T. K., Jiang, X., Freeh, V. W. and Liang, Z.: Jump-oriented programming: a new class of code-reuse attack, *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ASIACCS 2011, Hong Kong, China, March 22-24, 2011* (Cheung, B. S. N., Hui, L. C. K., Sandhu, R. S. and Wong, D. S., eds.), ACM, pp. 30–40 (online), DOI: 10.1145/1966913.1966919 (2011).

[4] Schuster, F., Tendyck, T., Liebchen, C., Davi, L., Sadeghi, A. and Holz, T.: Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications, *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, IEEE Computer Society, pp. 745–762 (online), DOI: 10.1109/SP.2015.51 (2015).

[5] Shacham, H.: The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86), *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007* (Ning, P., di Vimercati, S. D. C. and Syverson, P. F., eds.), ACM, pp. 552–561 (online), DOI: 10.1145/1315245.1315313 (2007).

[6] Roemer, R., Buchanan, E., Shacham, H. and Savage, S.: Return-Oriented Programming: Systems, Languages, and Applications, *ACM Trans. Inf. Syst. Secur.*, Vol. 15, No. 1, pp. 2:1–2:34 (online), DOI: 10.1145/2133375.2133377 (2012).

[7] Abadi, M., Budiu, M., Erlingsson, Ú. and Ligatti, J.: Control-flow integrity, *Proceedings of the 12th ACM Conference on Computer and Communications Security, CCS 2005, Alexandria, VA, USA, November 7-11, 2005* (Atluri, V., Meadows, C. A. and Juels, A., eds.), ACM, pp. 340–353 (online), DOI: 10.1145/1102120.1102165 (2005).

[8] : Control-flow Enforcement Technology Specification, https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf.

[9] Tran, M., Etheridge, M., Bletsch, T. K., Jiang, X., Freeh, V. W. and Ning, P.: On the Expressiveness of Return-into-libc Attacks, *Recent Advances in Intrusion Detection - 14th International Symposium, RAID 2011, Menlo Park, CA, USA, September 20-21, 2011. Proceedings* (Sommer, R., Balzarotti, D. and Maier, G., eds.), Lecture Notes in Computer Science, Vol. 6961, Springer, pp. 121–141 (online), DOI: 10.1007/978-3-642-23644-0_7 (2011).

[10] Quinlan, D. and Liao, C.: The ROSE source-to-source compiler infrastructure, *Cetus users and compiler infrastructure workshop, in conjunction with PACT*, Vol. 2011, Citeseer, p. 1 (2011).

[11] : LevelDB, https://github.com/google/leveldb.

[12] : GCC, the GNU Compiler Collection, https://gcc.gnu.org/.
[13] : The LLVM Compiler Infrastructure, https://llvm.org/.
[14] Pawlowski, A., Contag, M., van der Veen, V., Ouwehand, C., Holz, T., Bos, H., Athanasopoulos, E. and Giuffrida, C.: MARX: Uncovering Class Hierarchies in C++ Programs, *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*, The Internet Society, (online), available from ⟨https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/marx-uncovering-class-hierarchies-c-programs/⟩ (2017).
[15] Prakash, A., Hu, X. and Yin, H.: vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries, *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, The Internet Society, (online), available from ⟨https://www.ndss-symposium.org/ndss2015/vfguard-strict-protection-virtual-function-calls-cots-c-binaries⟩ (2015).
[16] Agadakos, I., Jin, D., Williams-King, D., Kemerlis, V. P. and Portokalidis, G.: Nibbler: debloating binary shared libraries, *Proceedings of the 35th Annual Computer Security Applications Conference, ACSAC 2019, San Juan, PR, USA, December 09-13, 2019* (Balenson, D., ed.), ACM, pp. 70–83 (online), DOI: 10.1145/3359789.3359823 (2019).
[17] Hu, H., Qian, C., Yagemann, C., Chung, S. P. H., Harris, W. R., Kim, T. and Lee, W.: Enforcing Unique Code Target Property for Control-Flow Integrity, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018* (Lie, D., Mannan, M., Backes, M. and Wang, X., eds.), ACM, pp. 1470–1486 (online), DOI: 10.1145/3243734.3243797 (2018).
[18] Zhang, C., Song, D., Carr, S. A., Payer, M., Li, T., Ding, Y. and Song, C.: VTrust: Regaining Trust on Virtual Calls, *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*, The Internet Society, (online), available from ⟨http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/vtrust-regaining-trust-virtual-calls.pdf⟩ (2016).
[19] van der Veen, V., Göktas, E., Contag, M., Pawlowski, A., Chen, X., Rawat, S., Bos, H., Holz, T., Athanasopoulos, E. and Giuffrida, C.: A Tough Call: Mitigating Advanced Code-Reuse Attacks at the Binary Level, *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, IEEE Computer Society, pp. 934–953 (online), DOI: 10.1109/SP.2016.60 (2016).
[20] Pawlowski, A., van der Veen, V., Andriesse, D., van der Kouwe, E., Holz, T., Giuffrida, C. and Bos, H.: VPS: excavating high-level C++ constructs from low-level binaries to protect dynamic dispatching, *Proceedings of the 35th Annual Computer Security Applications Conference, ACSAC 2019, San Juan, PR, USA, December 09-13, 2019* (Balenson, D., ed.), ACM, pp. 97–112 (online), DOI: 10.1145/3359789.3359797 (2019).
[21] DeLozier, C., Lakshminarayanan, K., Pokam, G. and Devietti, J.: Hurdle: Securing Jump Instructions Against Code Reuse Attacks, *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020 [ASPLOS 2020 was canceled because of COVID-19]* (Larus, J. R., Ceze, L. and Strauss, K., eds.), ACM, pp. 653–666 (online), DOI: 10.1145/3373376.3378506 (2020).