Computer Security Symposium 2020
26 - 29 October 2020

# バイナリ解析に基づく仮想デバイスへの不正 I/O 要求を防ぐフィルタの自動生成

庄司 豊[1,a)]　石黒 健太[1,b)]　河野 健二[1,c)]

**概要**：ハイパーバイザはクラウド環境の基盤であり，その脆弱性を突く攻撃を未然に防止する必要がある．ハイパーバイザにおけるデバイスのエミュレーションは，その仕様の複雑さから脆弱性の温床となっている．実際，QEMU というエミュレータでは 2016 年以降，104 件の脆弱性（CVSS v2 スコア 4.0 以上）が報告されており，そのうち 47 件（45.2%）がデバイスエミュレーションにおける脆弱性である．本論文では，デバイスエミュレーションの脆弱性を突く攻撃の多くが，デバイスドライバからの正当な I/O 要求シーケンスとは異なるものになっていることに着目し，デバイスドライバが生成し得ない I/O 要求のシーケンスを遮断する I/O 要求フィルタを実現する．デバイスドライバのバイナリコードを静的解析し，そのデバイスドライバの生成しうる I/O 要求のシーケンスを取得する．I/O 要求フィルタはデバイスドライバと仮想デバイスの間で動作し，事前に取得した I/O 要求シーケンスに合致しない I/O 要求を遮断する．この手法を Linux 5.2.0 の IDE および FDC に適用し，各デバイスに対する攻撃である CVE-2018-0959 および VENOM が防御できることを示す．

# Automatic Creation of Filters that Reject Illegal I/O Requests to Virtual Devices via Binary Analysis

Yutaka Shoji[1,a)]　Kenta Ishiguro[1,b)]　Kenji Kono[1,c)]

***Abstract:*** A hypervisor is indispensable for cloud environments. To guarantee strict isolation of virtual machines, it is mandatory to defend against attacks that exploit vulnerabilities in hypervisors. Unfortunately, device emulation in the hypervisor is a hotbed of vulnerabilities because of the complexity of emulated devices. In QEMU, a widely used emulator, 104 vulnerabilities, whose CVSS v2 score is 4.0 or higher, are reported since 2016 and 47 out of them (45.2%) are related to the device emulation. In this paper, we introduce an I/O request filter that filters out invalid sequences of I/O requests that device drivers never issue. The key insight behind the I/O request filter is that the I/O request sequences issued to exploit vulnerabilities usually differ from legal ones issued by device drivers. We automatically extract valid sequences of I/O requests from binary code of guest device drivers. The I/O request filter runs between guest device drivers and virtual devices and blocks I/O requests that do not conform to the I/O request sequences extracted in advance. The I/O request filter has been implemented on Linux 5.2.0 and QEMU 2.9.50, and we demonstrate it can defend against CVE-2018-0959 for IDE and VENOM for FDC.

## 1. はじめに

今日のマルチテナント型のクラウドにおいて，効率的な資源の活用および仮想マシン（VM）間の分離の観点から，仮想化は重要な技術となっている．仮想化によって，複数の VM がひとつの物理マシン上で隔離されて動作する．ハイパーバイザは，仮想デバイスが複数存在するように見せるため，デバイスエミュレーションにより物理ハードウェアを多重化し，複数の VM からのアクセスを調停する．

ハイパーバイザにおいて，デバイスエミュレーションは古くから脆弱性の温床となっている．これは，物理ハードウェアの仮想化は複雑で，デバイスの内部仕様まで理解

[1] 慶應義塾大学
Keio University
[a)] yutakashoji@sslab.ics.keio.ac.jp
[b)] kentaishiguro@sslab.ics.keio.ac.jp
[c)] kono@sslab.ics.keio.ac.jp

する必要があることによる．例えば，2009 年には Cloudburst [13] という VMware におけるビデオデバイスの脆弱性が，2015 年には VENOM [8] という QEMU の仮想フロッピーディスクコントローラ（FDC）の脆弱性が報告された．National Vulnerability Database (NVD) [14] によると，2016 年以降では Common Vulnerability Scoring System (CVSS) v2 スコアが 4.0 以上の深刻な脆弱性 104 件のうち，45% 以上の 47 件がデバイスエミュレーションに関係する．

本論文では，デバイスエミュレーションにおける脆弱性のエクスプロイトを防ぐ I/O 要求フィルタを提案する．I/O 要求フィルタは，フィルタ生成機構によってゲスト環境にインストールされたデバイスドライバのバイナリから静的に自動生成され， VM とハイパーバイザの間で動作する．デバイスドライバのバイナリの利用は，脆弱性を悪用するために発行される I/O 要求が，正当なデバイスドライバが発行する要求とは異なることが多いという観点に基づく．なぜなら，デバイスドライバが生成する I/O 要求に対して，デバイスが不正な挙動を示す場合，開発段階でのドライバの修正が期待されるためである．

I/O 要求フィルタを生成する機構は，デバイスドライバのバイナリを解析し，デバイスドライバが発行しうる一連の I/O 要求を静的に抽出する．そして，抽出した一連の I/O 要求からフィルタを自動で生成する．この一連の I/O 要求，すなわち，どのような I/O 要求がどの順番で発行されるかを本論文では **I/O シーケンス** と呼ぶ．抽出された **I/O シーケンス**に適合しない I/O 要求は不正な I/O 要求とみなし，実行時にフィルタアウトする．

I/O 要求フィルタの静的な生成は，カーネルイメージが生成された後かつ，クラウドにデプロイされる前の，マルウェアに感染していない安全な状況下で実行されることを想定する．さらに，クラウドの管理者は，信頼できる機関からカーネルのソースコードを取得し，悪意のあるコードを含まないカーネルイメージを用意するものとする．

I/O 要求フィルタの有効性を示すためにフィルタを生成する機構を実装し，複数の実験を行った．対象は QEMU+KVM [4], [12] であり，ポート I/O 方式を扱った．フィルタは型安全な言語 Rust [21] で記述され，フィルタ全体のセキュリティを向上させる．解析およびフィルタの生成は，Intel Corporation 82078 FDC [10] および Intel Corporation 82371SB PIIX3 IDE [11] に対して行った．実験では，VENOM および CVE-2018-0959 の proof-of-concept (PoC) コードを用い，I/O 要求フィルタが仮想デバイスおける脆弱性のエクスプロイトを拒否できることを確認した．さらに, IDE のフィルタについて filebench [2] を用いて性能を測定し，最大で 24% のスループット低下および 57% の遅延増大が発生することを確認した．

本論文は次のように構成される．第 2 章ではデバイスエミュレーションの脆弱性をひとつ取り上げる．第 3 章では利用する脅威モデルについて説明する．第 4 章で **I/O シーケンス**の抽出について説明し，第 5 章で解析機構およびフィルタの実装について述べる．第 6 章では自動生成された I/O 要求フィルタの実験結果を報告する．第 7 章では関連研究に言及する．最後に第 8 章で結論を述べる．

## 2. デバイスエミュレーションにおける脆弱性

仮想デバイスの脆弱性を悪用する攻撃では，デバイスドライバからは決して発行されない **I/O シーケンス**が発行される．この観点から CVE-2018-0959 を取り上げる．CVE-2018-0959 は hyper-V における IDE エミュレータの脆弱性であり，CVSS v2 スコアは 7.4 と深刻な脆弱性となっている [14]．この脆弱性の悪用により，攻撃者はリモートコード実行が可能となる．

この脆弱性の原因は，OS と IDE ドライブ間でのデータのやりとりに利用されるバッファへアクセスする際に，境界チェックが欠如しており，IDE エミュレータがゲストに範囲外アクセスを許してしまうことにある．

Hyper-V の IDE エミュレータでは，*IDE_DRIVE_STATE* 構造体が IDE ドライブの状態を管理する．この構造体は *TrackCacheBuffer* という問題のバッファへのポインタをメンバに持つ．そして，バッファにおいて次の読み書きに利用されるアドレスを指定するために，*Saved* というメンバが持つ *DriveStateBufferOffset* と *CurrentByte* という変数が利用される．

IDE ドライブ内のデータにアクセスする際，ゲストが IDE コントローラのデータレジスタに対応する I/O ポートアドレスにアクセスする．その際に *ReadDataPort* と *WriteDataPort* 関数を利用する．これらの関数内では，*TrackCacheBuffer*, *TrackCacheBufferOffset*, *CurrentByte* の 3 変数を利用して，バッファ内のアドレスを計算し，その結果を実際のデータ転送を担う *RltCopyMemory* 関数に渡す．しかし，*RltCopyMemory* 関数に渡されるアドレスが，*TrackCacheBuffer* が指すバッファの範囲内に存在するかを確認しない．したがって，*TrackCacheBufferOffset* を不正に大きな値にすることで，攻撃者は *ReadDataPort* や *WriteDataPort* 関数を悪用して範囲外へアクセスできる．

*TrackCacheBufferOffset* に大きな値をセットすることは IDE コントローラのコマンドレジスタに値を書き込むことで実現できる．Hyper-V のコードは非公開であるため，コマンドレジスタへのアクセス時に呼ばれる関数の詳細は不明であるものの，Bialek によればコマンドレジスタへのアクセスによって *DriveStateBufferOffset* の値を 0x200 ずつ増大させることが可能である [5].

コード 1 は，CVE-2018-0959 の PoC コードである．このコードでは, 9-10 行目で 0x1f7 に連続で書き込み *DriveStateBufferOffset* の値を不当に大きくする．そして，12 行

**Listing 1** CVE-2018-0959 の PoC コード [5]

```
1  /* Put the device in the desired state */
2  WriteIoPort(0x1F3, 0x10, 1);
3  WriteIoPort(0x1F2, 0x77, 1);
4  WriteIoPort(0x1F7, 0x30, 1);
5  for (DWORD bytesWritten = 0; bytesWritten < 0
       x200; bytesWritten += 4)
6      WriteIoPort(0x1F0, 0x41414141, 4);
7  /* Each write increments the
       DriveStateBufferOffset by 0x200 */
8  /* Make it huge! */
9  for (DWORD i = 0; i < 0x10000; i++)
10     WriteIoPort(0x1F7, 0x30, 1);
11 /* Trigger a write to the TrackCacheBuffer */
12 WriteIoPort(0x1F0, 0x13371337, 4);
13 /* Trigger a read from TrackCacheBuffer */
14 LeakedData = ReadIoPort(0x1F0, 4);
```

**Listing 2** Linux の IDE ドライバコードの一例 [1]

```
1  static ide_startstop_t do_reset1 (ide_drive_t
       *drive, int do_not_try_atapi) {
2      ide_hwif_t *hwif = drive->hwif;
3      const struct ide_tp_ops *tp_ops = hwif->
           tp_ops;
4      if (...) {
5          outb(drive->select | ATA_DEVICE_OBS,
               hwif->io_ports.device_addr);
6          udelay(20);
7          outb(ATA_CMD_DEV_RESET, hwif->io_ports.
               command_addr);
8      }
9  }
```

目および 14 行目で，*WriteDataPort* 関数や *ReadDataPort* 関数を呼び出すトリガを引き，範囲外へのアクセスを実現する．このコードの根幹は，コマンドレジスタの I/O ポートアドレスである 0x1f7 へ連続して値を書き込むことにある．これによって *DriveStateBufferOffset* を大きな値とし，範囲外アクセスを引き起こす．

デバイスドライバが発行する通常の **I/O シーケンス**では，コマンドレジスタへの書き込みは連続では行われない．つまり，コマンドレジスタへの書き込みの前後には他のデバイスレジスタへのアクセスが存在する．例えば，コード 2 は，Linux の IDE コントローラのデバイスドライバの抜粋であり，IDE コントローラのコマンドレジスタにアクセスする [1]．この関数では，7 行目のコマンドレジスタへの書き込み前に，5 行目でドライブ/ヘッドレジスタへの書き込みが行われる．このように， PoC コードの **I/O シーケンス**とデバイスドライバが発行するような通常の **I/O シーケンス**とを区別することが可能である．

## 3. 脅威モデル

本論文では，デバイスエミュレーションにおける脆弱性の悪用からハイパーバイザを保護することを目標とする．攻撃者は，VM 内で特権を持ち，ゲスト環境から任意の特権命令を実行できることを想定する．そのため不正な **I/O シーケンス**を発行し，仮想デバイスの脆弱性をエクスプロイトし，ハイパーバイザおよび同一ホスト上の他の VM への攻撃を行うことが可能である．

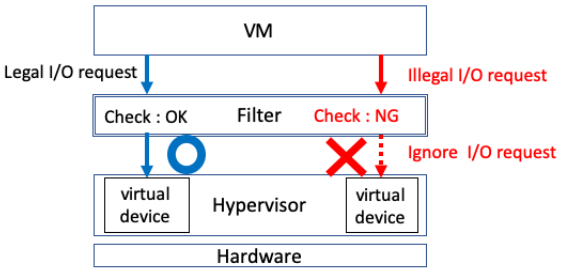



**図 1** I/O 要求フィルタの概要
**Fig. 1** Architectural Overview of an I/O Request Filter

クラウドプロバイダはカーネルイメージおよびフィルタをユーザに提供する．フィルタは，クラウドにデプロイするカーネルイメージのビルド後かつデプロイ前に生成する．これは，一度イメージがデプロイされると，悪意あるユーザによってカーネルのイメージが改変される可能性があるためである．カーネルイメージには不正なコードは含まれず，プロバイダには悪意はないことを前提とする．

## 4. I/O シーケンスの抽出

本章では，まず第 4.1 節で I/O 要求フィルタの概要について述べる．そして，第 4.2 節でリサーチカーネルである xv6 [6] を用いて抽出される **I/O シーケンス**の具体像を提供した後，バイナリから **I/O シーケンス**を抽出する手法について説明する．第 4.3 節では，Linux での適用事例について述べる．また，以降では x86_64 のポート I/O 方式について扱うが，他のアーキテクチャやメモリマップド I/O (MMIO) や Direct Memory Access (DMA) などの他の I/O 方式についても同様に適用できる．

### 4.1 I/O 要求フィルタ概要

**図 1** は，ゲスト VM からの I/O 要求の正当性を確認する I/O 要求フィルタの概要である．フィルタはハイパーバイザと VM との間に存在し，VM Exit 時に発生する VM からハイパーバイザへの遷移時に介入する．もし VM が発行した I/O 要求が フィルタが持つ **I/O シーケンス**にそぐわない場合，不正な要求として拒否する．

### 4.2 バイナリ解析による I/O シーケンス抽出

I/O 要求フィルタは，デバイスドライバのバイナリ解析を通して抽出された **I/O シーケンス**から生成される．解析および生成は自動で行われる．第 2 章で述べたように，

**Listing 3** xv6 [7] の IDE ドライバのコード抜粋

```
1  // Start the request for b. Caller must hold
       idelock.
2  static void idestart(struct buf *b)
3  {
4    int sector_per_block = BSIZE/SECTOR_SIZE;
5    int sector = b->blockno * sector_per_block;
6    int read_cmd = (sector_per_block == 1) ?
         IDE_CMD_READ : IDE_CMD_RDMUL;
7    int write_cmd = (sector_per_block == 1) ?
         IDE_CMD_WRITE : IDE_CMD_WRMUL;
8    idewait(0);
9    outb(0x3f6, 0); // generate interrupt
10   outb(0x1f2, sector_per_block); // # of sectors
11   outb(0x1f3, sector & 0xff);
12   outb(0x1f4, (sector >> 8) & 0xff);
13   outb(0x1f5, (sector >> 16) & 0xff);
14   outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((sector
         >>24)&0x0f));
15   if(b->flags & B_DIRTY){
16     outb(0x1f7, write_cmd);
17     outsl(0x1f0, b->data, BSIZE/4);
18   } else {
19     outb(0x1f7, read_cmd);
20   }
21 }
22
23 // Wait for IDE disk to become ready.
24 static int idewait(int checkerr)
25 {
26   int r;
27   while(((r = inb(0x1f7)) & (IDE_BSY|IDE_DRDY))
28     != IDE_DRDY);
29   return 0;
30 }
```

攻撃者はドライバが決して発行し得ない一連の I/O 要求を発行することが多いため，**I/O シーケンス**によって攻撃を検知することは合理的である．

正当な **I/O シーケンス**は，デバイスドライバが発行し得る一連の I/O 要求であり，I/O 要求の種別，すなわち Read/Write に加えて，次の 3 点から特徴付けられる．

- アクセスするデバイスレジスタ
- Write 操作の際にデバイスレジスタに書き込まれる値
- Read 操作の際にデバイスレジスタから読まれた値の利用の有無

ここで，アクセスされるデバイスレジスタは I/O ポートアドレスによって識別できる．

コード 3 は xv6 の IDE ドライバの抜粋であり，**図 2** はコード 3 から抽出された **I/O シーケンスグラフ**である．**I/O シーケンスグラフ**は非決定性有限オートマトンの形をとる．したがって，I/O 要求の正当性の確認は，非決定性有限オートマトンを決定性有限オートマトンに変換することと同じ要領で，ある状態から他の状態への遷移が成功するか否かに対応する．各ノードはドライバが発行しうる I/O 要求を表し，I/O 要求の種別とアクセスされるデバイスレジスタの種類，さらにデバイスレジスタに書き込まれる値の情報を持つ．例えば 2 番のノードは，0x3f6 に 0x0 を書き込むという I/O 要求を表す．エッジはそれが指す先の I/O 要求が次に発行される可能性があることを表す．例えば，7 番のノードから出る 2 つのエッジから，次に 8 番ノード か 9 番ノードの I/O 要求が発行されることがわかる．1 番のノードから出るふたつのエッジのラベルは，エッジが指す I/O 要求が発行される条件を示す．この場合は，1 番のノードの次に，2 番のノードの要求はエッジに記述される条件が成立する時のみ発行されることを示す．

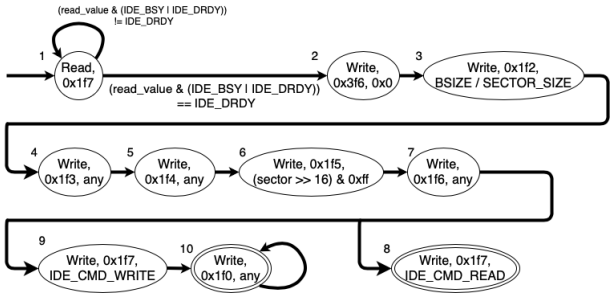



**図 2** xv6 のバイナリから抽出された **I/O シーケンスグラフ**

**Fig. 2** Extracted an I/O Sequence Graph from the xv6 Binary

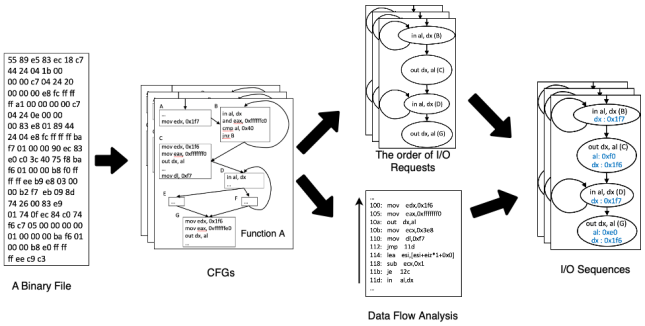



**図 3** バイナリ解析の概要

**Fig. 3** The Overview of Binary Analysis

I/O 要求フィルタの処理機構では，**I/O シーケンスグラフ**をフィルタとして扱う．I/O 要求が実行されたと推定されるノードの番号を保持し，ノードから出るエッジが指す全てのノードと，VM から発行された I/O 要求のマッチングを，エッジのラベルも考慮して行う．もし，一致するノードが存在しない場合は，当該要求を遮断する．

**I/O シーケンス**の抽出は，**図 3** に示すように 3 段階で行われる．まず最初にバイナリから各関数についてコントロールフローグラフ (CFG) を生成する．そして，I/O 命令の順番を取得する．同時に，取得された各 I/O 命令に対して，**I/O シーケンス**の 3 つの特徴を取得するためにデータフロー解析を行う．最後にこれらの結果を統合する．

データフロー解析は **I/O シーケンス**の 3 つの特徴に対して行う．I/O ポートアドレスおよびデバイスレジスタに書き込まれる値については，解析対象の I/O 命令について，これらの値が格納されるレジスタの use-def chain を

辿ることで値を特定する．これは，I/O ポートアドレスや，デバイスへのコマンドなどデバイスレジスタに書き込まれる値は，定数であり動的に計算されることはないという洞察に基づく．実際， xv6 の IDE ドライバでは， I/O ポートアドレスやコマンドが即値の形で代入されており容易に取得できる（コード 3）．Linux のような一般的な OS では，I/O ポートアドレスがデバイスを管理するためのデータ構造に格納されることがあるため，より複雑な解析が必要となる．これについては第 4.3 節で述べる．デバイスレジスタから読み出された値の利用については，対象の In 命令について，読み出された値が格納されるレジスタの def-use chain を辿り，読み出された値が `cmp` や `test` などの条件分岐に利用されているか否かを解析する．これは，読み出された値がエラーの有無やデバイスの状態確認に利用され，続くコントロールフローが変更される場合があるためである．デバイスレジスタから読み出された値は `cmp` や `test` といった条件分岐命令に，ビットマスクなどの処理を行った後に利用されることが多い．

データフローの解析に失敗した場合でも生成された I/O 要求フィルタは偽陽性を生じない．もし，I/O ポートアドレスの解析に失敗した場合は，その I/O 命令では任意の I/O ポートへのアクセスを許可する．デバイスレジスタに書き込まれる値についても同様である．例えば図 2 の 7 番のノードでは，0x1f6 への任意の値の書き込みを許可する．これは 7 番のノード中の “any” という語に表される．分岐に利用されるデバイスレジスタから読み出された値の解析に失敗した際は，条件分岐による I/O 要求のフィルタリングは不可能だが，ある I/O 命令から他の I/O 命令への遷移が，ドライバによって行われ得る遷移であることは保証できる．

この設計によって，I/O 要求フィルタが攻撃を見逃したり，悪意のないドライバの動作を妨害することがないようにする．ここで，本論文で提案する I/O 要求フィルタは，全ての攻撃を防ぐことは想定せず，ハイパーバイザの攻撃面の削減を目的として設計している．

### 4.3 Linux のバイナリ解析

実用 OS のドライバでは，バイナリ解析を利用して I/O ポートアドレスやデバイスレジスタに書き込まれる値を決定することは難しい．これは，これらの値が xv6 のように即値で定義されたり，定数マクロで定義されたりするのではなくデバイスの状態を管理するデータ構造に格納される場合があるためである．

この問題の対処法として，カーネルイメージをクラウドにデプロイする前にデバイスドライバを動作させ動的に取得する方法がある．デプロイ前に行うのは，デプロイ後だとカーネルイメージを信頼できないためである．

しかし，そのためにはデバイスの全ての I/O ポートやコマンドを利用するようなワークロードを作成する必要がある．なぜなら，フィルタの生成段階で，I/O ポートアドレスなどの情報を完全に網羅する必要があるためである．しかし，ドライバは例外的な処理などもあり，このようなワークロードの生成は難しい．

そこで，この手法と静的解析を組み合わせることで I/O ポート情報を取得する．Linux では I/O ポート情報は，デバイスドライバの初期化の段階でカーネルによって管理されるデータ構造に登録される．したがって，デバイスドライバは， I/O ポートアドレス情報を含む構造体のアドレスをベースレジスタに指定し，そこからオフセットを計算することで I/O ポートアドレスを取得する．例えば，`rbx` がベースアドレスであり，アクセス先のデバイスレジスタのオフセットが 0x30 の場合は，`mov rdx, qword ds:[rbx + 0x30]` のような形で計算し， `rdx` の値を利用して I/O ポートにアクセスする．

本提案手法では，第 4.2 節で述べたデータフロー解析を通して， I/O 要求に利用される I/O ポートアドレスを取得している命令を特定し，構造体のメンバへのアクセスのためのオフセットを取得する．得られたオフセットによってアクセスされるデバイスレジスタを決定できる．実際の I/O ポートアドレスは，クラウドにデプロイする前のカーネルイメージを実行し，動的に決定する．静的解析によってフィルタを生成するために必要なオフセットは取得済みのため，構造体のベースアドレスを取得できればすぐに全ての I/O ポートアドレスを決定できる．したがって，全ての I/O 要求を実行する包括的なワークロードを生成する必要はない．

## 5. 実装

本論文では, x86_64 上で動作する KVM+QEMU [4], [12] のポート I/O 方式を対象に，自動でデバイスドライバのバイナリを解析し，**I/O シーケンス**を抽出，さらに I/O 要求フィルタのプロトタイプを生成する機構を実装した．この機構はバイナリ解析ツール ROSE [19] を用いて C++ で実装した．

I/O 要求フィルタは Rust で記述される [21]. Rust は型安全な言語であり，バッファオーバーフローのような脆弱性やセグメンテーションフォルトの発生を抑制できる．さらに，コンパイル型言語でガベージコレクションがないため，実行時のオーバーヘッドの低減も可能である．

I/O 要求フィルタを VM と QEMU の間に挿入するために，Foreign Function Interface (FFI) を導入し，動的に I/O 要求フィルタを KVM+QEMU にリンクする．このために QEMU において I/O 要求が処理される際に FFI の関数が呼ばれるように QEMU に複数のフックを加えた．具体的には，*exec.c* の *address_space_rw* 関数に FFI の関数を呼び出すコードを付加した．

表 1　解析したバイナリの情報
**Table 1**　Information of the Analyzed Binaries

| File Name | # of Instructions | Size | # of Basic Blocks |
|---|---|---|---|
| ide-core.ko | 13329 | 6.16 MB | 3738 |
| floppy.ko | 9375 | 0.76 MB | 2320 |

I/O ポートアドレスおよびデバイスレジスタに書き込まれる値を決定するデータフロー解析は，対象の I/O 命令から遡って行われる．解析では，値を追跡中のレジスタが格納される解析対象リストが導入される．このリストは初期化の段階で，I/O 命令の I/O ポートアドレスや書き込まれる値が格納されるレジスタが加えられる．そして，対象の I/O 命令の前に実行される各命令に対して，解析対象リストに含まれるレジスタの値の伝播を検査し，リストを更新していく．レジスタに即値が代入されるなどして，リストが空になると初期化時に加えられたレジスタの値が判明したとして，解析を終了する．

検査は，命令の種類，命令のデスティネーションレジスタの種類およびそのアクセスサイズに着目して行われる．命令の種類は，デスティネーションレジスタの値を変更する命令か，そして変更する場合は，`mov` 命令かそれ以外の `add` のような命令かに着目する．これは，`add` 命令の場合は，ソースレジスタに加えてデスティネーションレジスタの内容も引き続き追跡する必要があるためである．デスティネーションレジスタの種類は，同じ種類のレジスタが解析対象リストに存在するかを確認する．もし存在する場合は，レジスタの値の伝播を追跡するために，ソースレジスタをリストに追加する．アクセスサイズについては，検査中の命令でアクセスされるサイズが，リスト中の同じ種類のレジスタのサイズより小さい場合は，残りの部分の解析を続行する必要があるため着目する．この検査を分岐を考慮した全てのパスについて行う．

デバイスレジスタから読み出された値の，条件分岐での利用解析でも，同様に解析対象リストの概念を用いて各命令に対して検査を行う，ただし，解析する命令は，対象の I/O 命令に続く命令となる．さらにリスト中のレジスタが `cmp` や `test` などの条件分岐命令で利用されるかを追加で検査し，これにより読まれた値がフローの変更に利用されるか否かを検知する．ここで，`cmp` や `test` で比較される値は多くの場合定数である．したがって，この定数を利用してラベルに反映する．それ以外の複雑なケースでは，解析失敗として先に述べたように次に発行される可能性のある全ての I/O 要求を許可する．

## 6. 実験

Linux カーネルの 5.2.0 の FDC と IDE に対して，バイナリ解析およびフィルタのプロトタイプの生成を行い，生成されたフィルタについてセキュリティおよび性能評価を行った．QEMU では IDE コントローラは Intel Corporation 82371SB PIIX3 IDE [11] が，FDC は Intel Corporation 82078 FDC [10] がデフォルトのデバイスとしてエミュレートされる．したがって，IDE に対しては ide-core.ko を，FDC に対しては floppy.ko を解析した．piix.ko には DMA 関連の I/O ポートアクセス以外の I/O 要求は存在しないため，解析およびフィルタ生成は行わなかった．また，解析に用いたカーネルはレガシーの IDE を使用するように設定を変更してビルドしたものを用いた．解析したファイルの情報を **表 1** に，実験環境を**表 2** に示す．

表 2　実験環境
**Table 2**　Experiment Environment

| | |
|---|---|
| Host System | Ubuntu 18.04.2 LTS (Bionic Beaver) |
| Linux Kernel Version | 4.15.0-76-generic |
| Host CPU | Intel Xeon CPU X5560 @ 2.80GHz * 8 logical processors |
| Host Memory | 32 GB |
| Host QEMU | Modified 2.9.50 |
| ROSE | 0.9.10.213 |
| Guest System | Ubuntu 18.04.3 LTS (Bionic Beaver) |
| Linux Kernel Version | 5.2.0 |
| Guest VCPU | 4 VCPUs (IDE), 1 VCPU (FDC) |
| Guest Memory | 8 GB |

表 3　`in` 命令の解析結果
**Table 3**　Result of Binary Analysis for `in` Instructions

| File Name | I/O Port Address | | Use of a Return Value | Total |
|---|---|---|---|---|
| | success | failed | | |
| ide-core.ko | 91 | 8 | 25 | 99 |
| floppy.ko | 11 | 0 | 6 | 11 |

表 4　`out` 命令の解析結果
**Table 4**　Result of Binary Analysis for `out` Instructions

| File Name | I/O Port Address | | Written Values | | Total |
|---|---|---|---|---|---|
| | success | failed | success | failed | |
| ide-core.ko | 56 | 11 | 17 | 50 | 67 |
| floppy.ko | 10 | 0 | 0 | 10 | 10 |

### 6.1 バイナリ解析結果

**表 3** は `in` 命令の解析結果であり，**表 4** は `out` 命令の解析結果である．ここで，それぞれのバイナリファイルに含まれる DMA 関連の I/O ポートアドレスアクセスを実行する関数については手動で除外した．バイナリ解析を通して，I/O ポートアドレスは 89.8% で特定に成功した．また，デバイスレジスタから読まれた値の 28.2% がその後のフローに影響を及ぼすことが判明した．さらに，書き込まれる値は 22.1% で特定できた．

### 6.2 セキュリティ評価

I/O 要求フィルタが適切に動作することを確認する，す

なわち，I/O 要求フィルタは如何なる通常の操作も妨げることなく不正な I/O 要求を検知，拒否することを確認するために，生成された IDE および FDC のフィルタのプロトタイプに対して下記のような操作を行った．

まず，I/O 要求フィルタが正当な操作を妨げないことを確認するために，次の操作を VM 内で行った．

(1) ディスクをマウントする．
(2) 512 KB のファイルをディスクに書き込む．
(3) 512 KB のファイルを追記する．
(4) ファイルを読み出す．
(5) ファイルを削除する．
(6) ディスクをアンマウントする．

ハードディスクドライブは 1GB の容量のものを，フロッピーディスクは 2.8MB の容量のものを利用した．

I/O 要求フィルタは，これらの操作によって発行される I/O 要求を正しく許可し，誤検知は発生しなかった．IDE については第 6.3 節で，性能評価のために Filebench [2] の 3 つのワークロードを実行したが，この実行中にも誤検知は発生しなかった．

次に I/O 要求フィルタが不正な **I/O シーケンス**を検知できることを確認するために，PoC コードを実行して不正な **I/O シーケンス**を発行した．FDC については，oss-security のメーリングリストで記述されている VENOM の PoC コードを実行した [17]. IDE については，マイクロソフトによって発表された CVE-2018-0959 に対する PoC コードを実行した [5]. どちらの PoC コードも，I/O 要求フィルタによって検知，拒否された．

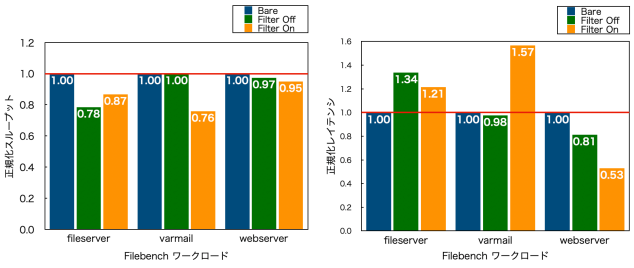
### 6.3 性能評価

I/O 要求フィルタによる性能への影響を確認するために，Filebench [2] のバージョン Filebench-1.5-alpha3 を利用し，IDE を対象にスループットと遅延を測定した．Filebench はファイルシステムやストレージのための様々なワークロードを生成するベンチマークである．具体的には fileserver, varmail, webserver の 3 つのワークロードを初期設定のまま 10 回実行し，その平均値を正規化して比較した．比較は Bare, Filter Off, Filter On の 3 つの条件下で行った．Bare では，I/O 要求フィルタをリンクせず VM を実行した．他の設定では I/O 要求フィルタをリンクして VM を実行した．ただし Filter Off では，フィルタリングを行わず，Filter On ではフィルタリングを行った．

**図 4** は，各ワークロードの正規化されたスループットと遅延を示す．スループットについては最大 24% の低下，遅延については最大 57% の増加があることを確認した．

## 7. 関連研究

デバイスエミュレーションはハイパーバイザにおける主要な攻撃ベクタのひとつである．Perez-Botero らによれば．デバイスエミュレーションは Xen の脆弱性の 33.9%. KVM の脆弱性の 39.5% を占める [18]．デバイスエミュレータにおける脆弱性の削減は，ハイパーバイザのセキュリティを向上するために一般的に提案されている．




(a) 正規化スループット　　(b) 正規化レイテンシ

**図 4** IDE 用の I/O 要求フィルタの性能オーバーヘッド
**Fig. 4** Performance Overhead of I/O Request Filters for IDE

ゲスト VM からの不正な I/O 要求はハイパーバイザのセキュリティを攻撃できる．Nioh [16] は，デバイスの仕様書を用いて，不正な I/O 要求による攻撃を阻止するデバイス要求フィルタを導入する．これはデバイスエミュレータの脆弱性への攻撃を試みる加工された I/O 要求は，デバイスの仕様から逸脱する傾向にあるためである．しかし，Nioh ではフィルタは手動で生成する必要がある．仕様書の曖昧さや，デバイスの挙動の複雑さを考慮すると，手動で生成する労力が必要となる．

ハイパーバイザの複雑さはセキュリティの無力化の一因である．なぜなら，複雑化しコードの絶対量が増大すると，より脆弱性が増加する可能性があるためである．そこでハイパーバイザの Trusted Computing Base (TCB) を削減し，攻撃面を削減する手法も提案されている．Min-V [15] は，クラウド環境で必要のないデバイスを無効化することで，ハイパーバイザのデバイスエミュレーションのコード量を削減する．

ハイパーバイザのバグを発見し，脆弱性を削減する手法も提案されている．Virtual CPU Validation [3] では，物理 CPU のテスト環境を仮想 CPU にも適用し，バグを検知する．Intel のテスト機構を KVM に適用し 117 のバグを発見し，そのうち 2 つは脆弱性であった．

バグを探索する手法としてファジングも有効である．VDF [9] では，コードの分岐をファジングするようなシードを用いることで，仮想デバイスにおける効率的なファジングを実現する．HYPER-CUBE [20] はファジング用の軽量カスタム OS を導入し，効率的で正確なファジングを実現している．

## 8. おわりに

ハイパーバイザは仮想化技術を実現するための鍵となる要素である．しかし，ハイパーバイザには VENOM のような重大な脆弱性が数多く存在する．デバイスエミュレー

ションは，その実装の難しさからハイパーバイザの脆弱性の主な原因となっている．本論文では，仮想デバイスの脆弱性への攻撃からハイパーバイザを保護することを目的とし，I/O 要求フィルタを提案した．その際，攻撃者はデバイスドライバが発行するような通常の方法とは異なる方法でデバイスを挙動させることに着目した．I/O 要求フィルタは，デバイスドライバのバイナリ解析を通して自動生成される．バイナリ解析では，I/O 要求の順序，種別，利用される I/O ポートアドレス，デバイスレジスタに書き込まれる値，読み出し命令で取得した値によるコントロールフローの変更の有無によって特徴付けられる **I/O シーケンス**を抽出する．生成した IDE および FDC のフィルタは，通常の挙動を妨げることなく PoC コードによる攻撃の阻止に成功した．性能評価では，IDE のフィルタについて，スループットが最大 24% 低下し，遅延が最大 57% 増加することが確認された．

**謝辞** 本研究は JSPS KAKENHI Grant Number JP19K11906 の助成を受けたものである．

## 参考文献

[1] : linux/ide-eh.c at v5.2・torvalds/linux, `https://github.com/torvalds/linux/blob/v5.2/drivers/ide/ide-eh.c`.

[2] : filebench/filebench: File system and storage benchmark that uses a custom language to generate a large variety of workloads., `https://github.com/filebench/filebench` (2020).

[3] Amit, N., Tsafrir, D., Schuster, A., Ayoub, A. and Shlomo, E.: Virtual CPU Validation, *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, New York, NY, USA, Association for Computing Machinery, p. 311–327 (online), DOI: 10.1145/2815400.2815420 (2015).

[4] Bellard, F.: QEMU, a Fast and Portable Dynamic Translator, *Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*, USENIX, pp. 41–46 (online), available from ⟨http://www.usenix.org/events/usenix05/tech/freenix/bellard.html⟩ (2005).

[5] Bialek, J.: Exploiting the Hyper-V IDE Emulator to Escape the Virtual Machine, BlackHat USA '19 (2019).

[6] Cox, R., Kaashoek, F. and Morris, R.: Xv6, a simple Unix-like teaching operating system, `https://pdos.csail.mit.edu/6.828/2017/xv6.html` (2017).

[7] Cox, R., Kaashoek, F. and Morris, R.: xv6_code_ref, `https://pdos.csail.mit.edu/6.828/2017/xv6/xv6-rev10.pdf` (2017).

[8] CrowdStrike: VENOM Vulnerability, `https://venom.crowdstrike.com` (2015).

[9] Henderson, A., Yin, H., Jin, G., Han, H. and Deng, H.: VDF: Targeted Evolutionary Fuzz Testing of Virtual Devices, *Research in Attacks, Intrusions, and Defenses - 20th International Symposium, RAID 2017, Atlanta, GA, USA, September 18-20, 2017, Proceedings*, pp. 3–25 (online), DOI: 10.1007/978-3-319-66332-6_1 (2017).

[10] Intel Corporation: 82078 44 PIN CHMOS SINGLE-CHIP FLOPPY DISK CONTROLLER (1995).

[11] Intel Corporation: 82371FB (PIIX) AND 82371SB (PIIX3) PCI ISA IDE XCELERATOR (1997).

[12] Kivity, A., Kamay, Y., Laor, D., Lublin, U. and Liguori, A.: KVM: the Linux Virtual Machine Monitor, *In Proceedings of the 2007 Ottawa Linux Symposium (OLS' 07)* (2007).

[13] Kortchinsky, K.: Cloudburst, BlackHat USA '09, (online), available from ⟨https://www.blackhat.com/presentations/bh-usa-09/KORTCHINSKY/BHUSA09-Kortchinsky-Cloudburst-SLIDES.pdf⟩ (2009).

[14] National Institute of Standards and Technology: National Vulnerability Database, `https://www.nvd.nist.gov`.

[15] Nguyen, A., Raj, H., Rayanchu, S., Saroiu, S. and Wolman, A.: Delusional Boot: Securing Hypervisors without Massive Re-Engineering, *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, New York, NY, USA, Association for Computing Machinery, p. 141–154 (online), DOI: 10.1145/2168836.2168851 (2012).

[16] Ogasawara, J. and Kono, K.: Nioh: Hardening The Hypervisor by Filtering Illegal I/O Requests to Virtual Devices, *Proceedings of the 33rd Annual Computer Security Applications Conference*, ACSAC 2017, New York, NY, USA, Association for Computing Machinery, p. 542–552 (online), DOI: 10.1145/3134600.3134648 (2017).

[17] oss security: 'Re: [oss-security] VENOM - CVE-2015-3456' - MARC, `https://marc.info/?l=oss-security\&m=143155206320935\&w=2` (2015).

[18] Perez-Botero, D., Szefer, J. and Lee, R. B.: Characterizing Hypervisor Vulnerabilities in Cloud Computing Servers, *Proceedings of the 2013 International Workshop on Security in Cloud Computing*, Cloud Computing '13, New York, NY, USA, Association for Computing Machinery, p. 3–10 (online), DOI: 10.1145/2484402.2484406 (2013).

[19] Quinlan, D. and Liao, C.: The ROSE Source-to-Source Compiler Infrastructure, Cetus Users and Compiler Infrastructure Workshop '11 (2011).

[20] Schumilo, S., Aschermann, C., Abbasi, A., Worner, S. and Holz, T.: HYPER-CUBE: High-Dimensional Hypervisor Fuzzing, *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, CA, USA, February 23-26, 2020, Proceedings*, (online), available from ⟨https://doi.org/10.14722/ndss.2020.23096⟩ (2020).

[21] The Rust Community: Rust Programming Language, `https://www.rust-lang.org` (2020).