論文紹介

PrIntFuzz: Fuzzing Linux Drivers via Automated Virtual Device Simulation

[ISSTA'22]

Z. Ma, B. Zhao, L. Ren, Z. Li, S. Ma, X. Luo, and C. Zhang, "Printfuzz: Fuzzing linux drivers via automated virtual device simulation," in Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, ser. ISSTA 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 404–416. https://doi.org/10.1145/3533767.3534226

PrIntFuzz [1] の概要

ジャンル： Linux デバイスドライバファジング

問題提起： ・既存のカーネルファザーはサポートするデバイスの **数・機能** が少ない

→ コードカバレッジの低下

提案手法： ・静的解析を利用した仮想デバイスの自動作成

・初期化処理へのソフトウェアフォルトの挿入

・システムコール・デバイス I/O ・割り込みといった複数の
入力ソースからの多次元ファジング

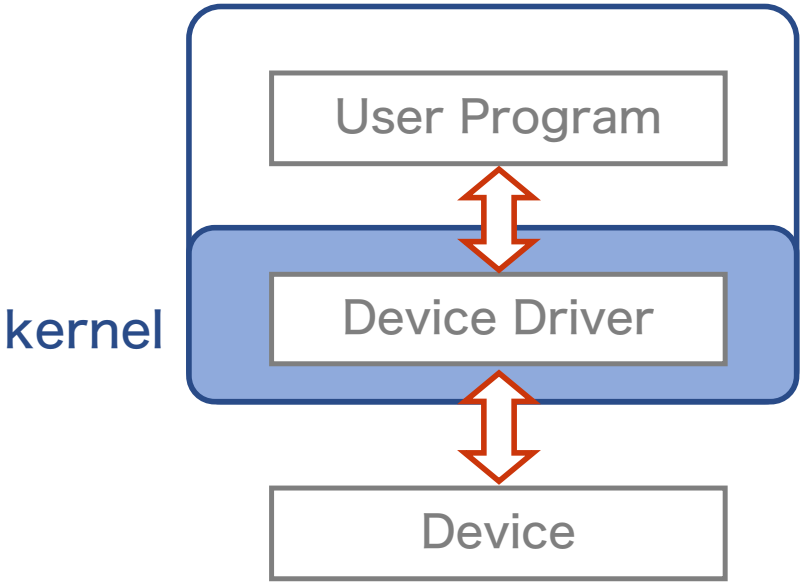
結果： ・311 / 472 / 169 個の PCI / I2C / USB デバイスを自動生成

・デバイスドライバの 150 個のバグを発見

[1] PrIntFuzz: fuzzing Linux drivers via automated virtual device simulation [ISSTA'22]

デバイスドライバ

・ハードウェアとソフトウェアで相互のやりとりを可能にするソフトウェア

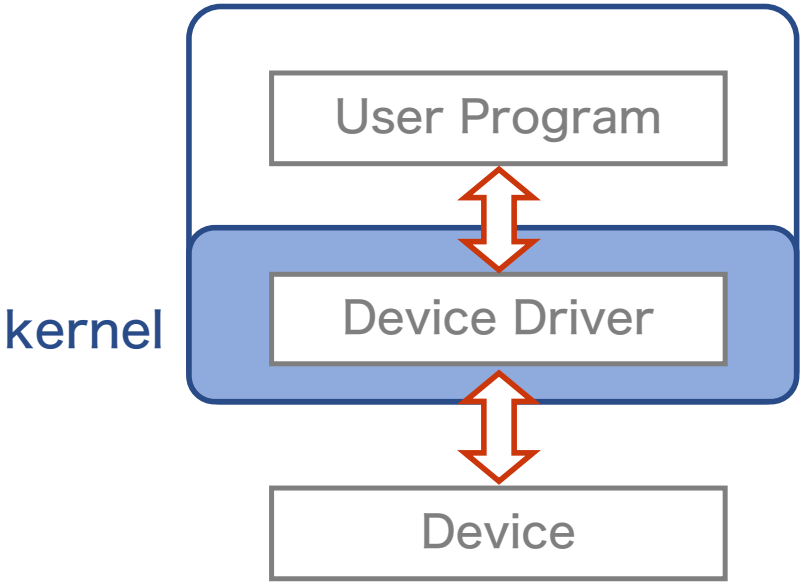


・Linux の脆弱性の多くはデバイスドライバ関連
  ・Jeff は Android の 85% の脆弱性がドライバ関連と指摘 [2]
  ・Arireza は Linux のほとんどの脆弱性がドライバ関連と指摘 [3]

[2] https://events.static.linuxfound.org/sites/events/files/slides/Android-%20protecting%20the%20kernel.pdf
[3] Understanding Linux kernel vulnerabilities. Journal of Computer Virology and Hacking Techniques (2021), 1-14

デバイスドライバ

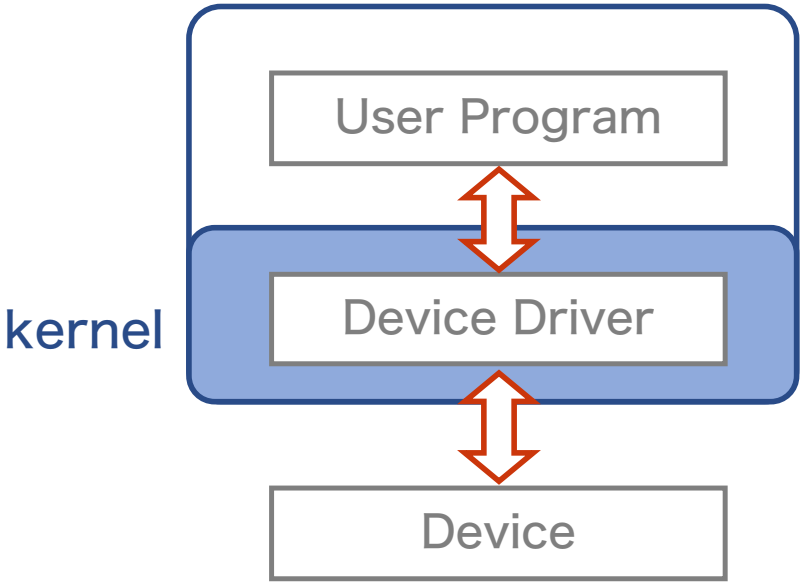
Q. デバイスドライバへの入力とは？



デバイスドライバ

Q. デバイスドライバへの入力とは？

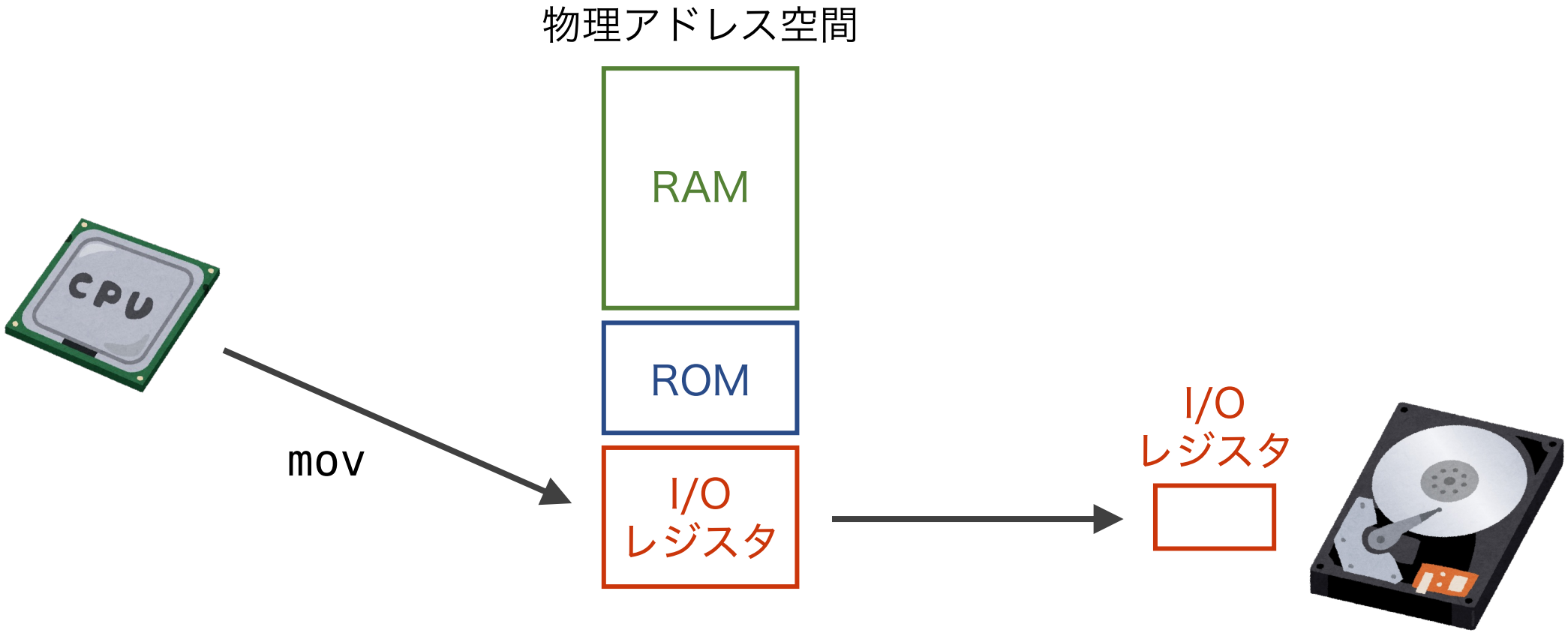
- ユーザ空間からの入力
  - システムコールの引数
- デバイスからの入力
  - MMIO（Memory-Mapped I/O）
  - PMIO（Port-Mapped I/O）
  - DMA（Direct Memory Access）
  - IRQ（Interrupt ReQuest）



MMIO (Memory-mapped I/O)

・デバイスのレジスタは物理アドレス空間にマップされる

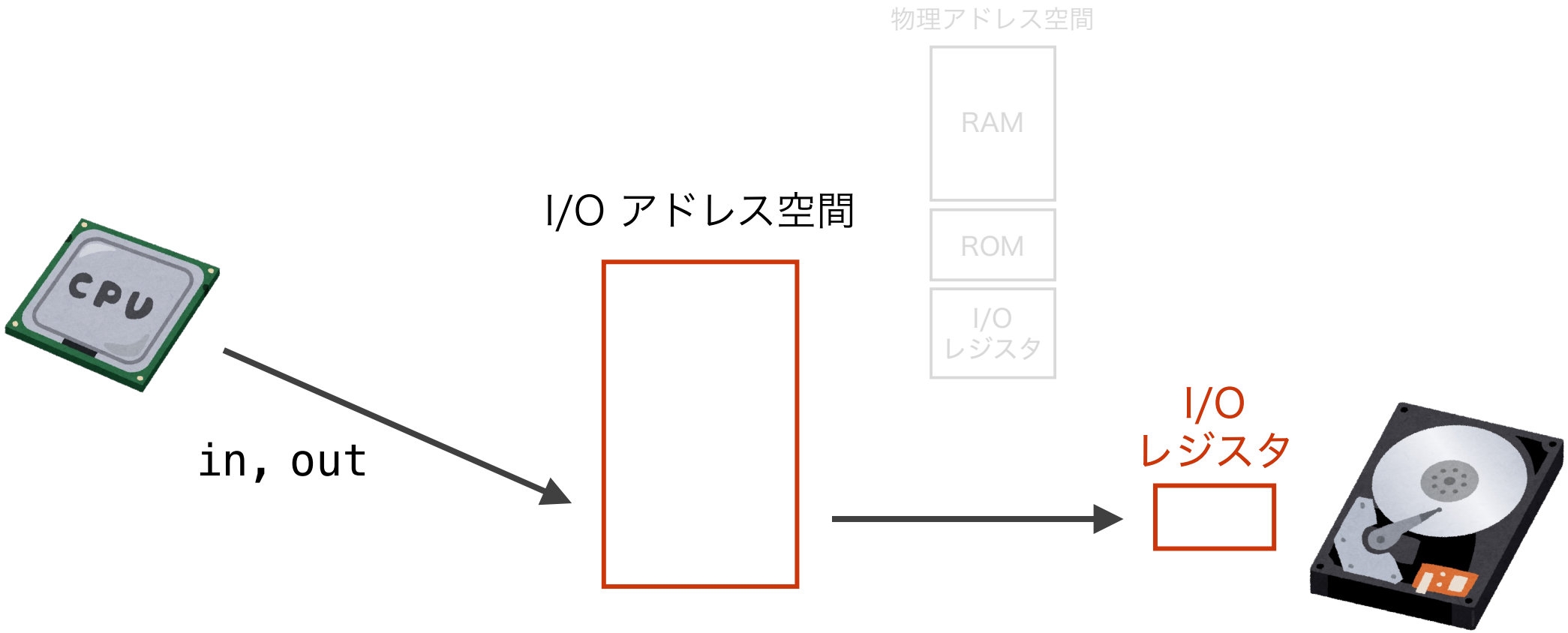
→ 通常のメモリアクセス命令（mov など）を使用してアクセス



PMIO (Port-mapped I/O)

・物理アドレス空間とは別の I/O アドレス空間にマップされる

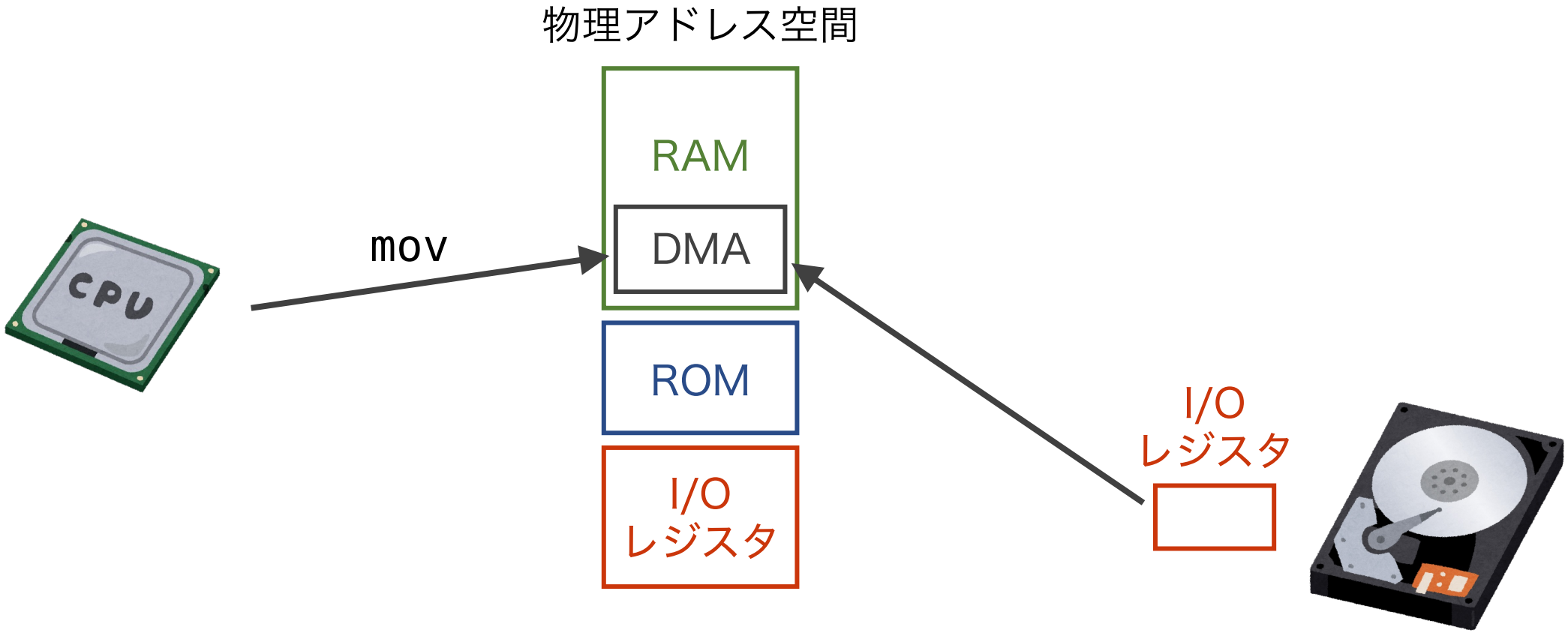
→ I/O 命令（`in, out` など）を使用してアクセス



DMA (Direct Memory Access)

・CPU を介さずにデバイスのデータをメインメモリと通信

→ 通常のメモリアクセス命令（mov など）を使用してアクセス



IRQ (Interrupt Request)

- デバイス側から CPU に特定のイベントを知らせる信号を送る

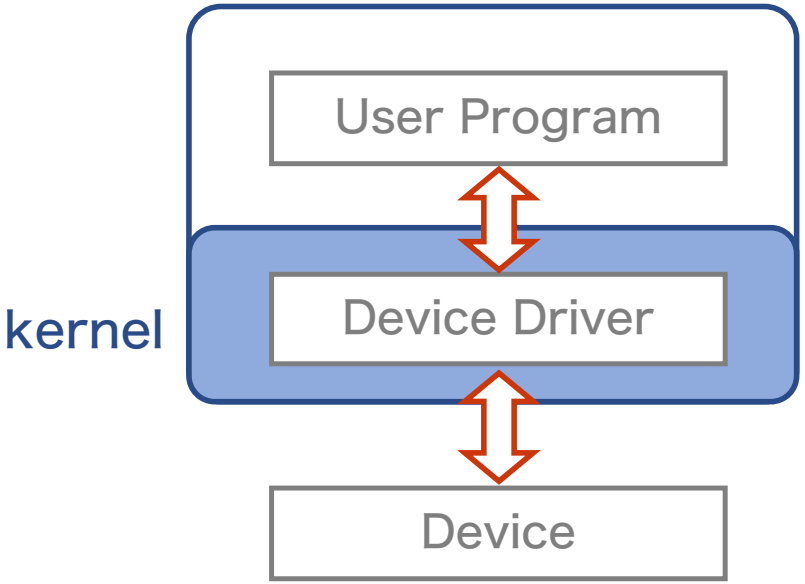
  → 処理完了, 新しいデータ到着 など

- CPU は現在の処理を中断し, 割り込みハンドラを実行する

デバイスドライバ

Q. デバイスドライバへの入力とは？

- ユーザ空間からの入力
  - システムコールの引数
- デバイスからの入力
  - MMIO（Memory-Mapped I/O）
  - PMIO（Port-Mapped I/O）
  - DMA（Direct Memory Access）
  - IRQ（Interrupt ReQuest）



⇨ デバイスドライバにはこれだけ多くの入力が存在

既存のデバドラファジングの問題点

・基本的にデバドラのファジングは実デバイスが必要

→ テストできる機会が限られ, ドライバのほとんどのコードをカバーできない

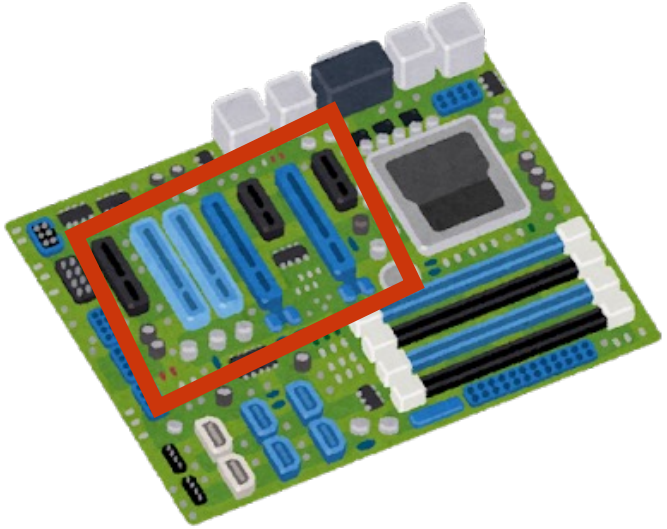
・最近では仮想デバイスを使用したファザーも

・USBFuzz [4] は USB デバイスをシミュレート

・VBA [5] は PCI デバイスをシミュレート

PCI : マザーボード上の拡張カードスロットを使用して
周辺機器と接続するインターフェース

ex.) グラフィックカード, ネットワークカード



[4] https://events.static.linuxfound.org/sites/events/files/slides/Android-%20protecting%20the%20kernel.pdf
[5] Understanding Linux kernel vulnerabilities. Journal of Computer Virology and Hacking Techniques (2021), 1-14

既存のデバドラファジングの問題点

- **問題点1**：サポートする仮想デバイスの **数** が限られる
  - USBFuzz, VBA はそれぞれ USB , PCI デバイスのみをシミュレート
  - これらは手動で作られた設定ファイルが必要

仮想デバイスを自動生成する必要あり

- **問題点2**：サポートする仮想デバイスの **機能** が限られる
  - USBFuzz は USB のプラグ, アンプラグ操作のみに集中

初期化・割り込み・I/O などデバイスの機能を
包括的にシミュレートする必要あり

提案手法

・静的解析を使用して, 仮想デバイスを自動作成

→ Probing (初期化処理) を通過できるようにする

・初期化処理にソフトウェアフォルトを挿入

→ 普段通過しないエラー処理をテスト

・デバイスドライバの機能をファザーによって探索

・システムコールからの入力

・割り込み注入

・デバイスへのデータの注入

→ 既存手法以上のコードカバレッジを実現する

PrIntFuzz

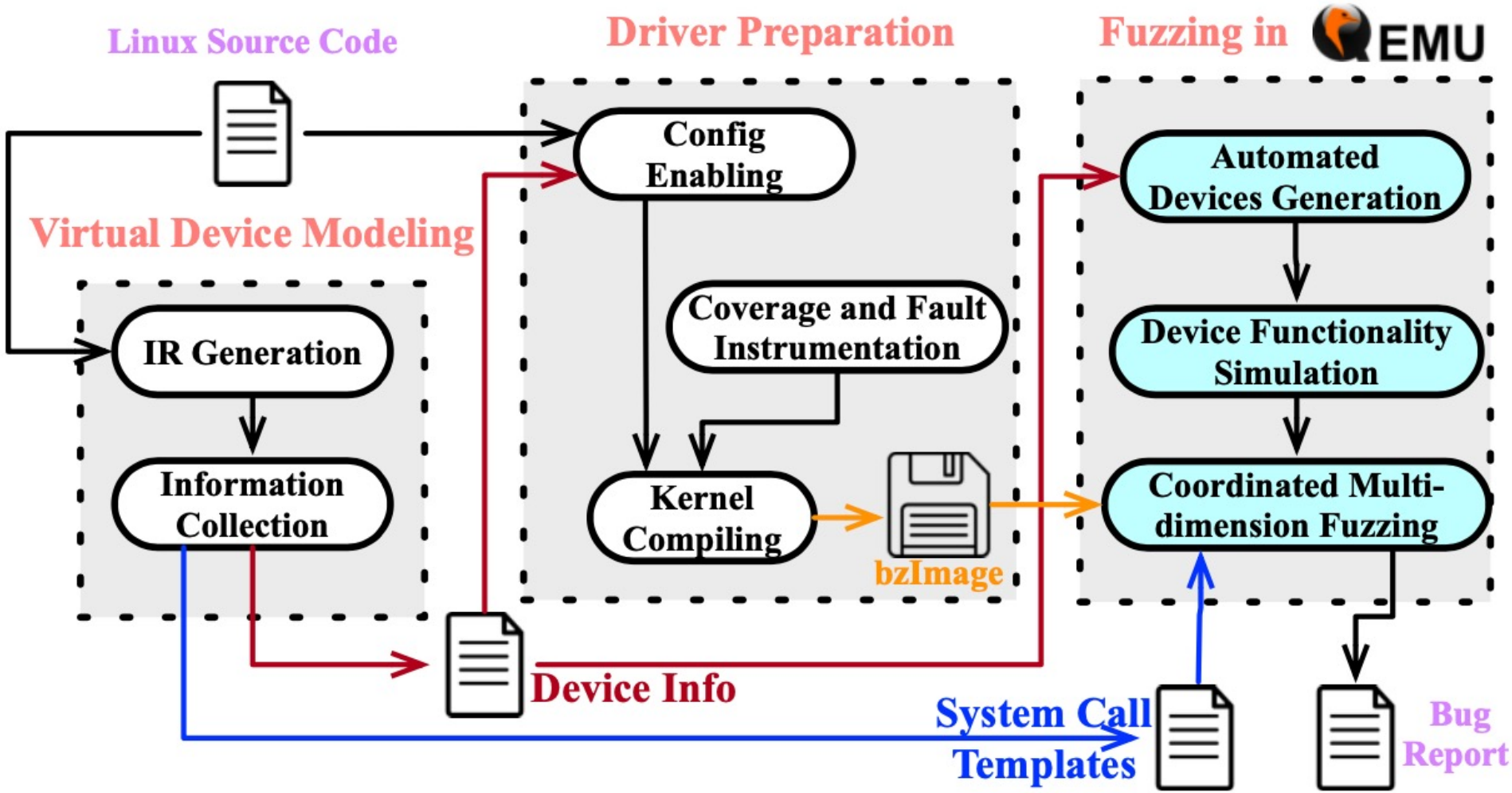


Figure 2: The framework of PrIntFuzz

PrIntFuzz

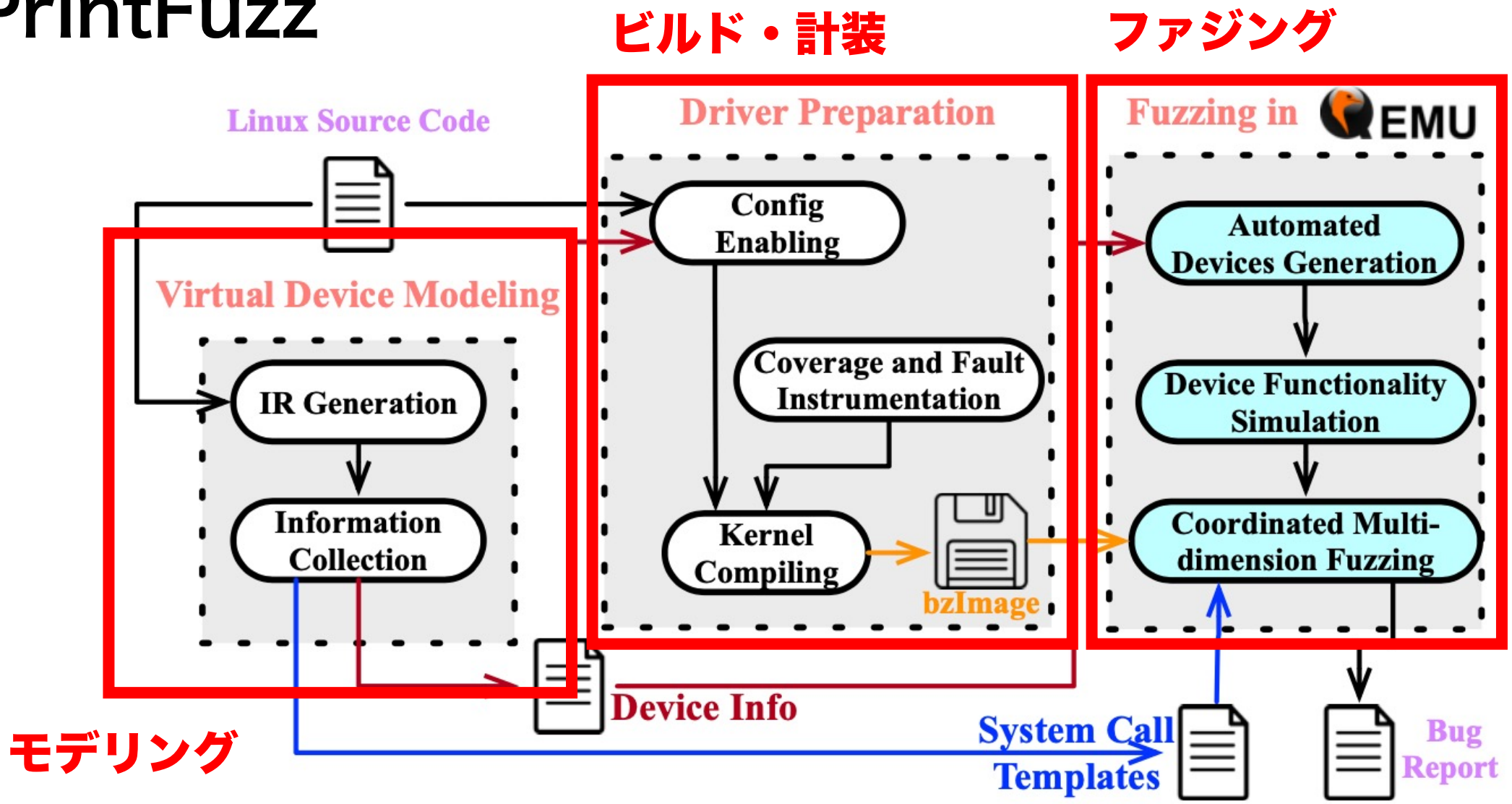


Figure 2: The framework of PrIntFuzz

PrIntFuzz

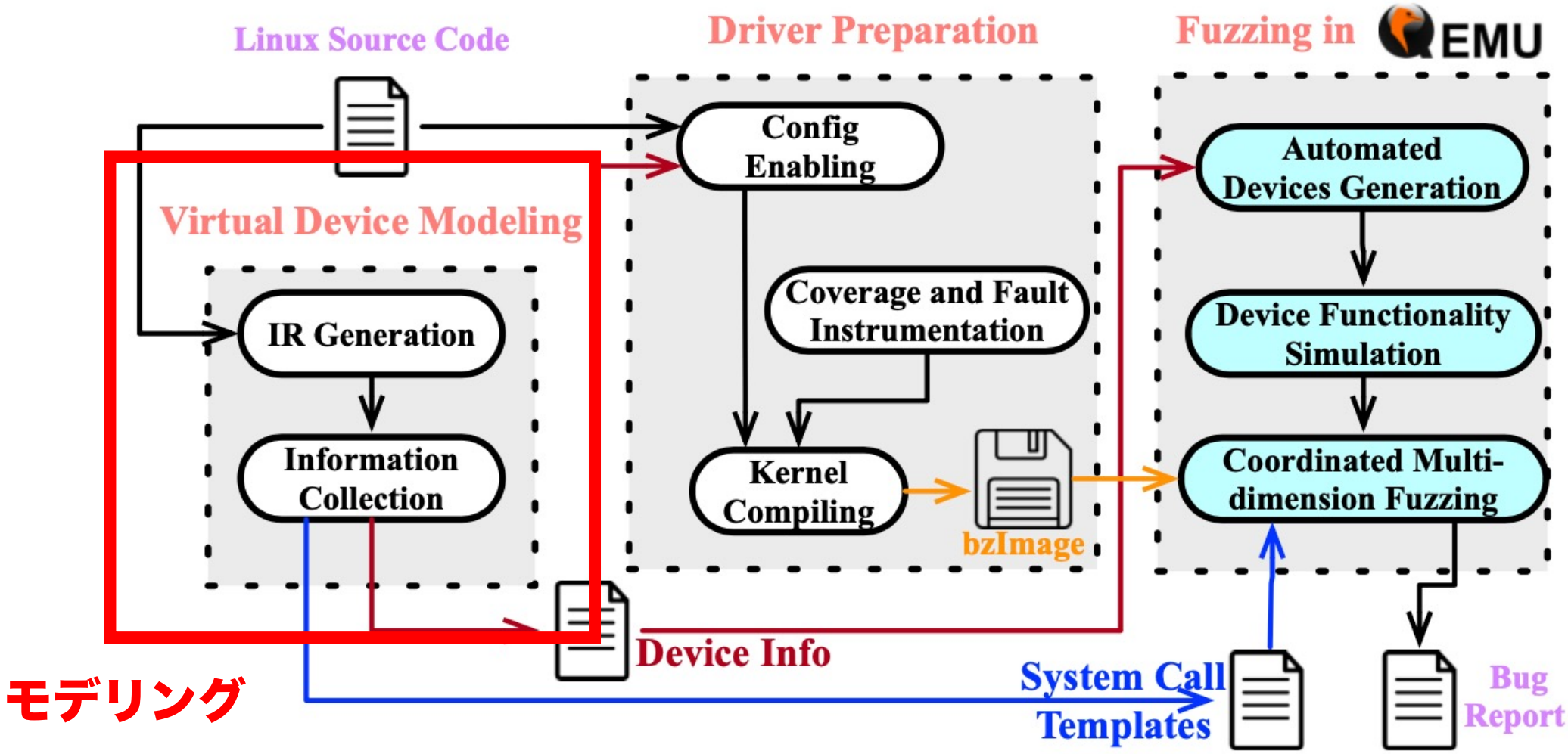


Figure 2: The framework of PrIntFuzz

モデリング

- 仮想モデリングの本質は probing（初期化処理）を通過させること
  - ドライバは初期化処理でデバイスのサニティチェックを行う
- 静的解析によってモデリングを行う
  - かなりヒューリスティックな解析
  - 多くの対応不可なケースが存在する
- ここからの説明は PCI デバイスの例
  - PCI デバイスはドライバ全体の 36% を占める
  - PrIntFuzz の手法は他のドライバにも適用可能

Data Space Modeling

・初期化処理でドライバはハードウェアレジスタから値を読み取り，
サニティチェックを行う

・ハードウェアからのデータが制約を満たさない場合エラーを起こす

```
1     status = he_readl(he_dev, RESET_CNTL);
2     if ((status & BOARD_RST_STATUS) == 0) {
3         hprintk("reset failed\n");
4         return -EINVAL;
5     }
```

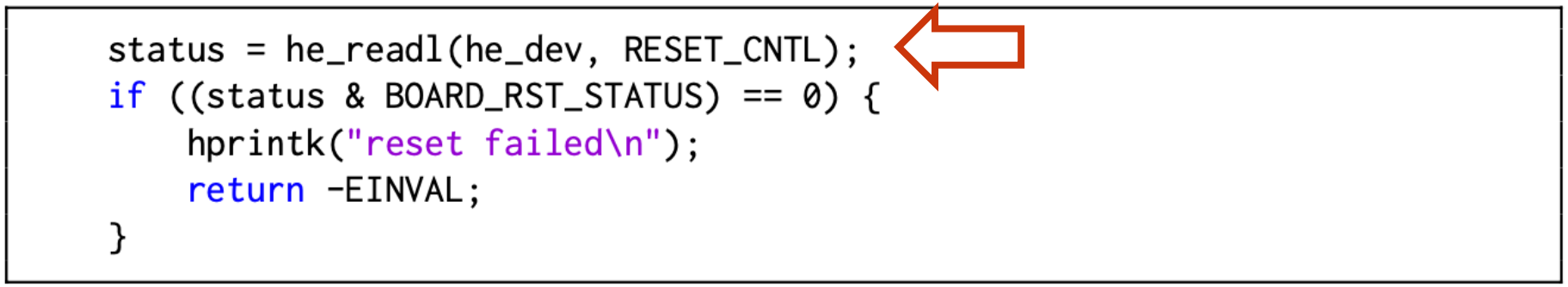
Listing 1: drivers/atm/he.c

Data Space Modeling

- Linux カーネルが提供するレジスタ読み取り関数を事前に設定

  ex.) `readl`, `inl`

- さらに, 読み取りのラッパ関数も識別する
  - 関数名に `read` が含まれる
  - 関数の基本ブロック数が 5 未満
  - この関数内でレジスタ読み取り関数が呼ばれる



```
1     status = he_readl(he_dev, RESET_CNTL);
2     if ((status & BOARD_RST_STATUS) == 0) {
3         hprintk("reset failed\n");
4         return -EINVAL;
5     }
```

Listing 1: drivers/atm/he.c

Data Space Modeling

- いくつかのパターンを用いてレジスタが持つべき値を推測する
  - 入力 → and でマスク → 定数と比較 （図のパターン）
  - 入力 → 定数と比較
  - 入力 → ビット拡張 → 定数と比較
- def-use chain をたどって, icmp 命令の使用を見つける
- icmp 命令の定数オペランドから, レジスタが持つべき値を設定

```
1     status = he_readl(he_dev, RESET_CNTL);
2     if ((status & BOARD_RST_STATUS) == 0) {
3         hprintk("reset failed\n");
4         return -EINVAL;
5     }
```

Listing 1: drivers/atm/he.c

Data Space Modeling

- いくつかのパターンを用いてレジスタが持つべき値を推測する
  - 入力 → and でマスク → 定数と比較 （図のパターン）
  - 入力 → 定数と比較
  - 入力 → ビット拡張 → 定数と比較
- def-use chain をたどって, icmp 命令の使用を見つける
- icmp 命令の定数オペランドから, レジスタが持つべき値を設定

```
1  MASK_BIT = 0x07;
2  status = readb(FOO_REG);
3  if ((status & MASK_BIT) != 0x03) {
4      // Fail
5      return -EINVAL;
6  }
```

Q. FOO_REG の値は何であるべき?

Data Space Modeling

- いくつかのパターンを用いてレジスタが持つべき値を推測する
  - 入力 → and でマスク → 定数と比較 （図のパターン）
  - 入力 → 定数と比較
  - 入力 → ビット拡張 → 定数と比較
- def-use chain をたどって, icmp 命令の使用を見つける
- icmp 命令の定数オペランドから, レジスタが持つべき値を設定

```
1  MASK_BIT = 0x0f;
2  status = readb(FOO_REG);
3  if ((status & MASK_BIT) != 0x03) {
4      // Fail
5      return -EINVAL;
6  }
```

Q. FOO_REG の値は何であるべき?

A. 0x03 など

（下位2ビットが立っていれば良い）

I/O and Memory Space Modeling

- 各 PCI デバイスは最大6つのメモリ・ I/O アドレス領域を設定可能
- ドライバはメモリ・ I/O アドレス領域のタイプをチェックし，
  一致しない場合はエラーを返す
- PrIntFuzz はこのチェックに関連するマクロや関数を事前に設定
  ex.) `pci_resource_flags`
- 領域の位置（`0`）とリソースタイプ（`IORESOURCE_IO`）を抽出

```
1     if (!(pci_resource_flags(dev, 0) & IORESOURCE_IO))
2         return -ENODEV;
```

Listing 2: drivers/i2c/buses/i2c-amd8111.c

Configuration Space Modeling

- 各 PCI デバイスはハードウェア情報を保持する, いくつかのコンフィグレーションレジスタを持つ
- 5つの標準レジスタの値を解析
  - ベンダー ID
  - デバイス ID
  - クラス
  - サブシステムベンダー ID
  - サブシステムデバイス ID

USB のベンダー ID [6]

Goeasily Int'l Co., Ltd.	8696
GoerTek Inc.	11517
GoFlight, Inc.	2547
GoHubs, inc.	2337
GOKO Imaging Devices Co., Ltd.	12845
Gold Cable (Zhongshan) Electronic Co., Ltd.	11813
Golden Bridge Electech Inc.	1680
Golden Bright (Sichuan) Electronic Technology Co Ltd	4370
Golden Emperor International Ltd.	9503
Golden Profit Electronics Ltd.	12939
Golden Transmart International Co., Ltd.	10451
Goldenconn Electronics Technology (Suzhou) Co., Ltd.	11320
Goldfinger	12767
Goldfull Electronics & Telecommunications Corp.	1842
Goldmund International	10218
Goldteck International Inc.	2558
Goldvish S.A.	7199
GOMETRICS, S.L.	9470
Gomez Sim Industries, LLC	13817
GONGNIU GROUP CO., LTD.	11864
Good Fancy Enterprise Co., Ltd.	5087
Good Man Corporation	3886
Good Mind Industries Co., Ltd.	4709
Good Technology, Inc.	3646
Good Way Technology Co., Ltd. & GWC technology Inc	1631
GOOD WILL Instrument Co., Ltd.	8580
Good Work Systems	4245
GOOD YEAR ELECTRONIC MFG. CO., LTD.	8306
GOODBETTERBEST Ltd.	9712
Goodong Industry Co., Ltd.	12882
GOOGFIT TECH LIMITED	12007
Google Inc.	6353
Goossens Engineering	8825
GOPEL electronic GmbH	2412
Gopod Group Limited	11521
Goppa, LLC	13120
GoPro	9842
Gosuncn RichLink Technology Co., Ltd.	14201
GOSUNCNWELINK TECHNOLOGY Co., LTD.	12378
GoTrustID Inc.	12963
Gould Instrument Systems	4493
Governors America Corp.	9369
Gowin Semicondcutor Corporation	13226
Gowin Technology International Holdings Limited	10678
Goyatek Technology Inc.	4631
GP Electronics (HK) Limited	10156
GPEG International	9468

[6] https://www.usb.org/sites/default/files/vendor_ids100824_0.pdf

Configuration Space Modeling

- コンフィグレーションレジスタは, デバイスをスキャンする際に読み取られ, デバイスがどのドライバと一致するか決定する
- ドライバ側では, `pci_device_id` 構造体によって定義
- PrIntFuzz はこの構造体から情報を取得する

```
1    struct pci_device_id {
2        __u32 vendor, device; /* Vendor and device ID or PCI_ANY_ID*/
3        __u32 subvendor, subdevice; /* Subsystem ID's or PCI_ANY_ID */
4        __u32 class, class_mask;  /* (class,subclass,prog-if) triplet */
5        kernel_ulong_t driver_data; /* Data private to the driver */
6    };
```

Listing 3: Device ID structure

Configuration Space Modeling

- さらに, ドライバは他の非標準レジスタをチェックすることもある
- PrIntFuzz には関連する関数を事前に与える

  ex.) `pci_read_config_xxx`

- これも同じ方法（def-use chain, `icmp`）でレジスタの取るべき値を解析

```
1     pci_read_config_dword(pdev, 0x80, &reg);
2     if (reg != ADM8211_SIG1 && reg != ADM8211_SIG2) {
3         printk("%s : Invalid signature (0x%x)\n", pci_name(pdev), reg);
4         err = -EINVAL;
5         goto err_disable_pdev;
6     }
```

Listing 4: drivers/net/wireless/admtek/adm8211.c

System Call Templetes Generation

- PrIntFuzz はドライバのインターフェースを解析して,
  自動的にシステムコールテンプレートを作成
  - Syzkaller でテストするには必要

```
syz_open_dev$floppy(dev ptr[in, string["/dev/fd#"]], id intptr, flags flags[fd_open_flags]) fd_floppy

ioctl$FLOPPY_FDEJECT(fd fd_floppy, cmd const[FDEJECT])
ioctl$FLOPPY_FDCLRPRM(fd fd_floppy, cmd const[FDCLRPRM])
ioctl$FLOPPY_FDSETPRM(fd fd_floppy, cmd const[FDSETPRM], arg ptr[in, floppy_struct])
ioctl$FLOPPY_FDDEFPRM(fd fd_floppy, cmd const[FDDEFPRM], arg ptr[in, floppy_struct])
ioctl$FLOPPY_FDGETPRM(fd fd_floppy, cmd const[FDGETPRM], arg ptr[out, floppy_struct])]
ioctl$FLOPPY_FDGETMAXERRS(fd fd_floppy, cmd const[FDGETMAXERRS], arg ptr[out, floppy_max_errors])
ioctl$FLOPPY_FDSETMAXERRS(fd fd_floppy, cmd const[FDSETMAXERRS], arg ptr[in, floppy_max_errors])
ioctl$FLOPPY_FDGETDRVTYP(fd fd_floppy, cmd const[FDGETDRVTYP], arg ptr[out, floppy_drive_name])
ioctl$FLOPPY_FDSETDRVPRM(fd fd_floppy, cmd const[FDSETDRVPRM], arg ptr[in, floppy_drive_params])
ioctl$FLOPPY_FDGETDRVPRM(fd fd_floppy, cmd const[FDGETDRVPRM], arg ptr[out, floppy_drive_params])
```

System Call Templetes Generation

- PrIntFuzz はドライバのインターフェースを解析して,
  自動的にシステムコールテンプレートを作成
- DIFuzz [7] を使用してドライバインターフェースからシステムコールを収集
- テスト対象のドライバインターフェースは特定の名前のついた
  (`operations`, `ops` など) 構造体から関数ポインタを解析して特定

```
1  struct file_operations syz_iquinux_fops = {
2      .owner = THIS_MODULE,
3      .open = syz_iquinux_open,
4      .unlocked_ioctl = syz_iquinux_ioctl,
5  };
```

[7] Difuzz : Interface aware fuzzing for kernel drivers [CCS'17]

PrIntFuzz

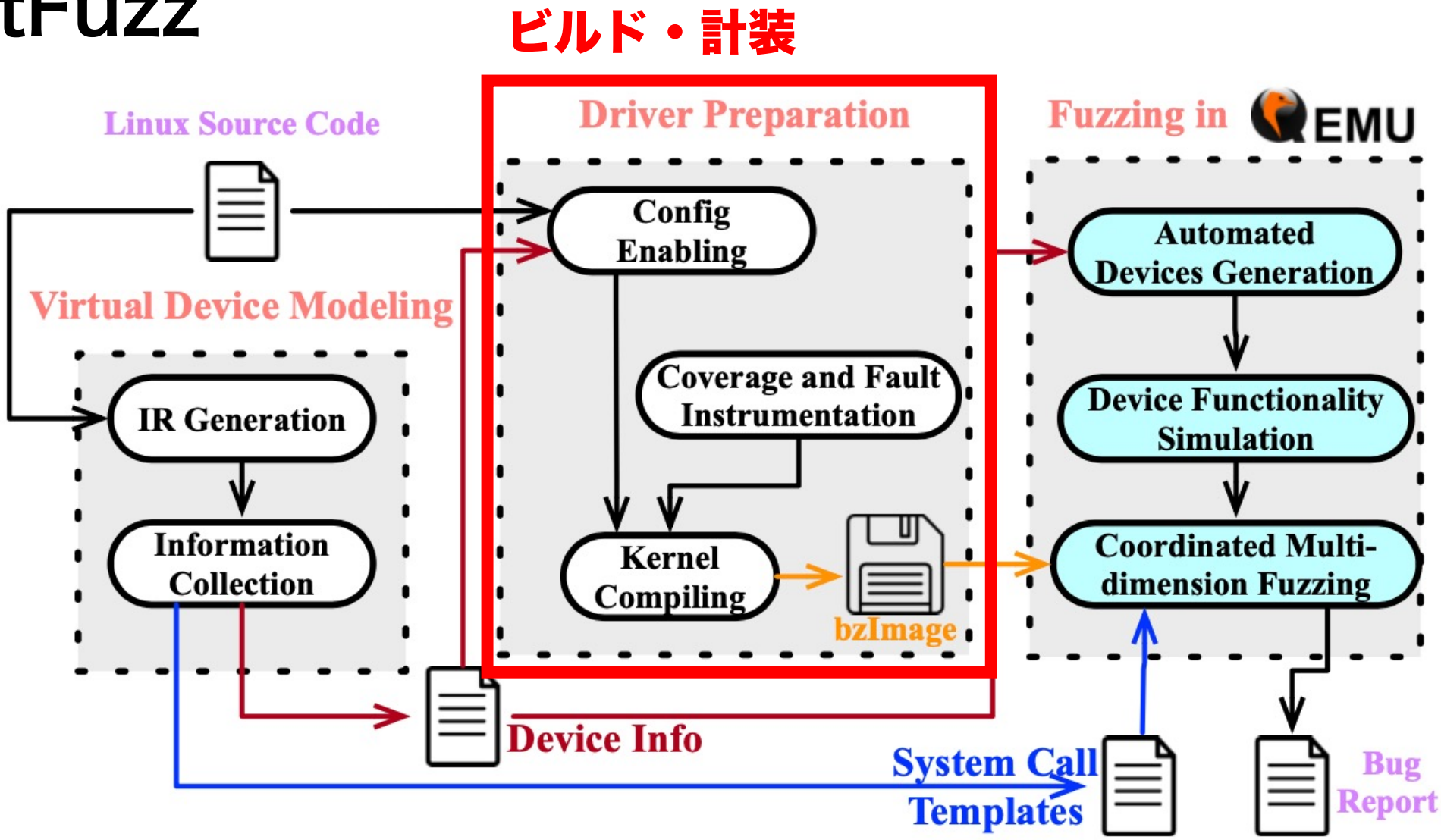


Figure 2: The framework of PrIntFuzz

Config Enabling

- ドライバをカーネルの設定で有効にする必要がある
- PrIntFuzz はドライバの設定名を含む Makefile を検索
- Kconfiglib を使用して Kconfig を解析し，ドライバが依存する
  他のカーネルオプションを見つける
- これらの設定を全て有効にする

Makefile

```
obj-$(CONFIG_DVB_DM1105) += dm1105.o
```

.config

```
CONFIG_DVB_DM1105=y
```

Kconfig

```
config DVB_DM1105
  depends on RC_CORE
```

[8] https://github.com/ulfalizer/Kconfiglib/

PrIntFuzz

ファジング

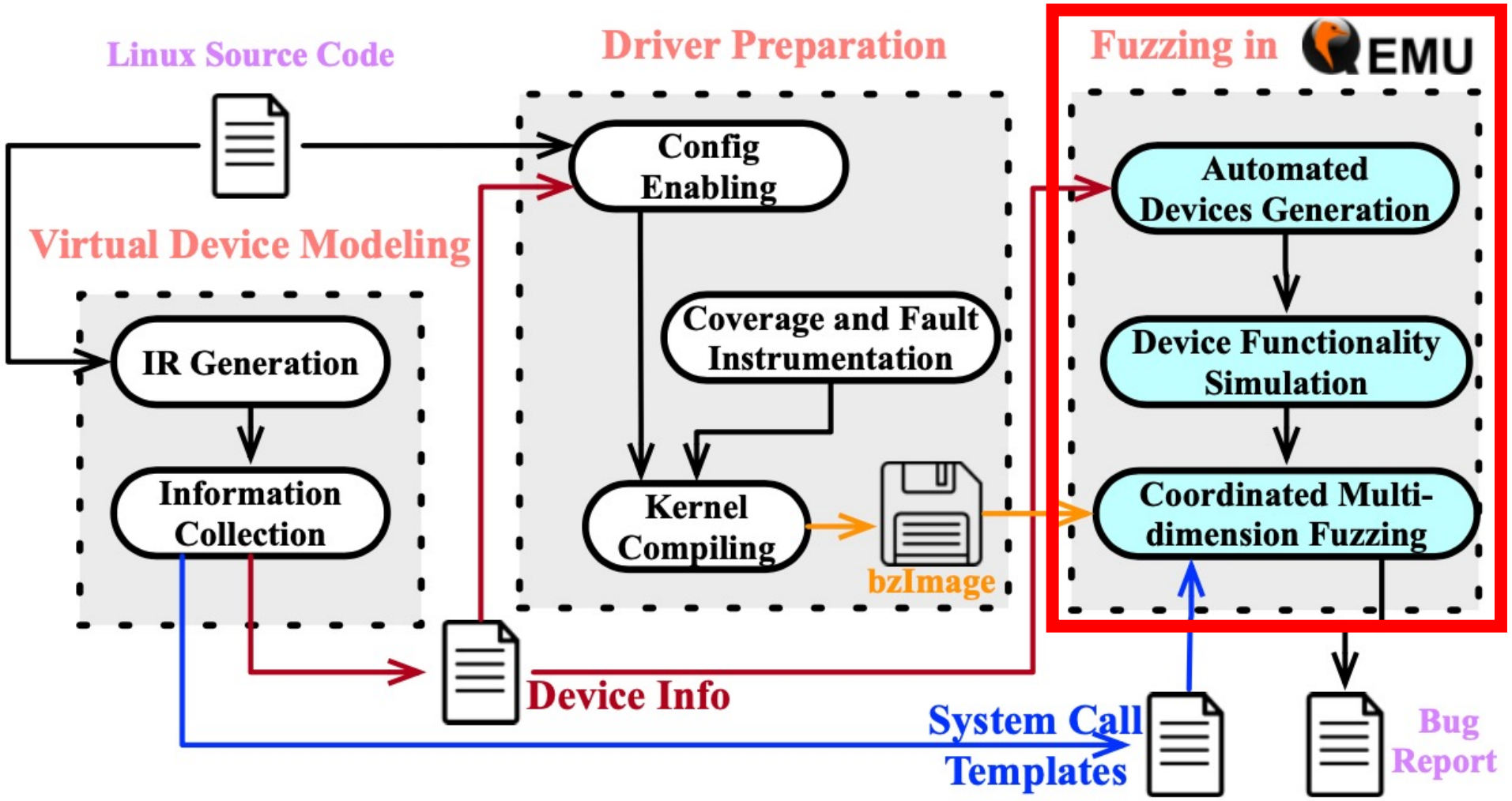


Figure 2: The framework of PrIntFuzz

Two Types of Fuzzing

・(論文にはまとめて書かれているが) 2種類のファジングがある

1. 初期化処理にソフトウェアフォルトを挿入するファジング

   → 普段通過しないエラー処理をテスト

2. デバイスドライバの機能を探索するファジング

   ・システムコールからの入力

   ・割り込み注入

   ・デバイスへのデータの注入

Fault Injection of the Initialization Code

- 初期化処理にソフトウェアフォルトを挿入するファジング
  → 普段通過しないエラー処理をテスト
- PrIntFuzz は以下の3つの条件を満たす関数をエラーサイトとみなす
  1. 関数の戻り値の型が整数かポインタ
  2. 関数の戻り値が条件文でチェックされる
  3. 関数が定義ではなく宣言
     → カーネル API に対してのみフォルト注入を行う

Fault Injection of the Initialization Code

- フォルトの注入は計装の段階で行う
- `fault_pos` はドライバのロード時に割り当てられる値
- `fault_pos` をファジングの各反復で異なる値に設定することで, 異なる関数で失敗することを可能にする

```
 1  static int curr_pos = 0;
 2  module_param(fault_pos, int, -1);
 3
 4  int foo_f1() {
 5      curr_pos++;
 6      if (fault_pos == curr_pos)
 7          return -1;
 8      return f1();
 9  }
10
11  int xxx_probe() {
12  -   if (f1()) {
13  +   if (foo_f1()) {
14          return -EINVAL;
15      }
16  }
```

元乃隅神社

Two Types of Fuzzing

・(論文にはまとめて書かれているが) 2種類のファジングがある

1. 初期化処理にソフトウェアフォルトを挿入するファジング

   → 普段通過しないエラー処理をテスト

2. デバイスドライバの機能を探索するファジング

   ・システムコールからの入力

   ・割り込み注入

   ・デバイスへのデータの注入

Automated Virtual Device Genaration

- 静的解析の結果から自動的に仮想デバイスを生成する
- 仮想デバイスは基本的なテンプレートに従って生成される

  → 大文字の部分に静的解析で得られた情報を埋め込む

```
1  void init(PCIDevice *pdev)
2  {
3      pdev->vendor_id = VENDOR_ID;
4      pdev->device_id = DEVICE_ID;
5      ...
6      pdev->buf = PCI_DATA;
7      pci_set_config(pdev, PCI_CNFIG_POS, PCI_CONFIG_VALUE);
8      pci_register_bar(pdev, PCI_REGION_POS, PCI_REGION_TYPE);
9      ...
10 }
```

Listing 5: Basic device template

Device Functionality Simulation

- 仮想デバイスはファジングに必要な最低限の機能のみを残す

  MMIO, PIO, DMA, 割り込み など

  → 例えば, ネットワークデバイスで本当にデータを送受信するようなことはしない

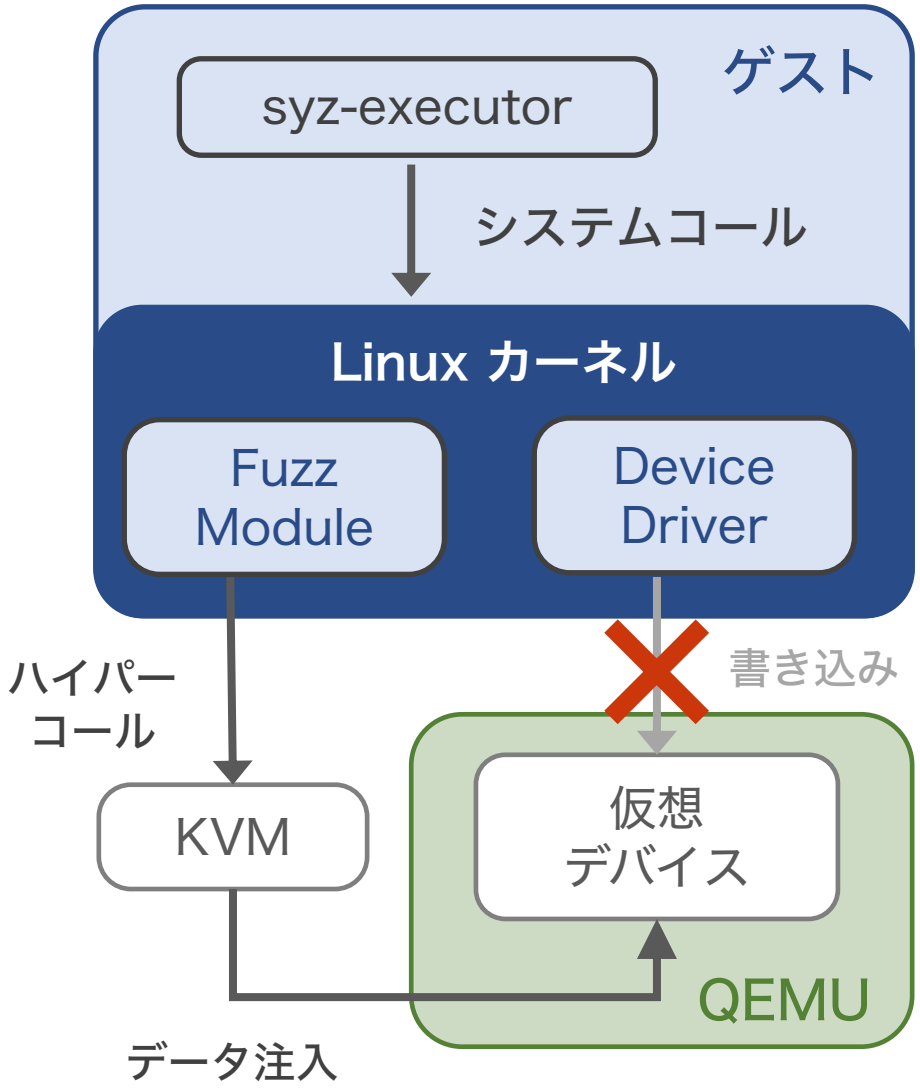
Multi-dimension Fuzzing

・ドライバからデバイスへの書き込み操作はすべて無視する

→ デバイスに書き込まれるのはファザーが注入するデータのみ

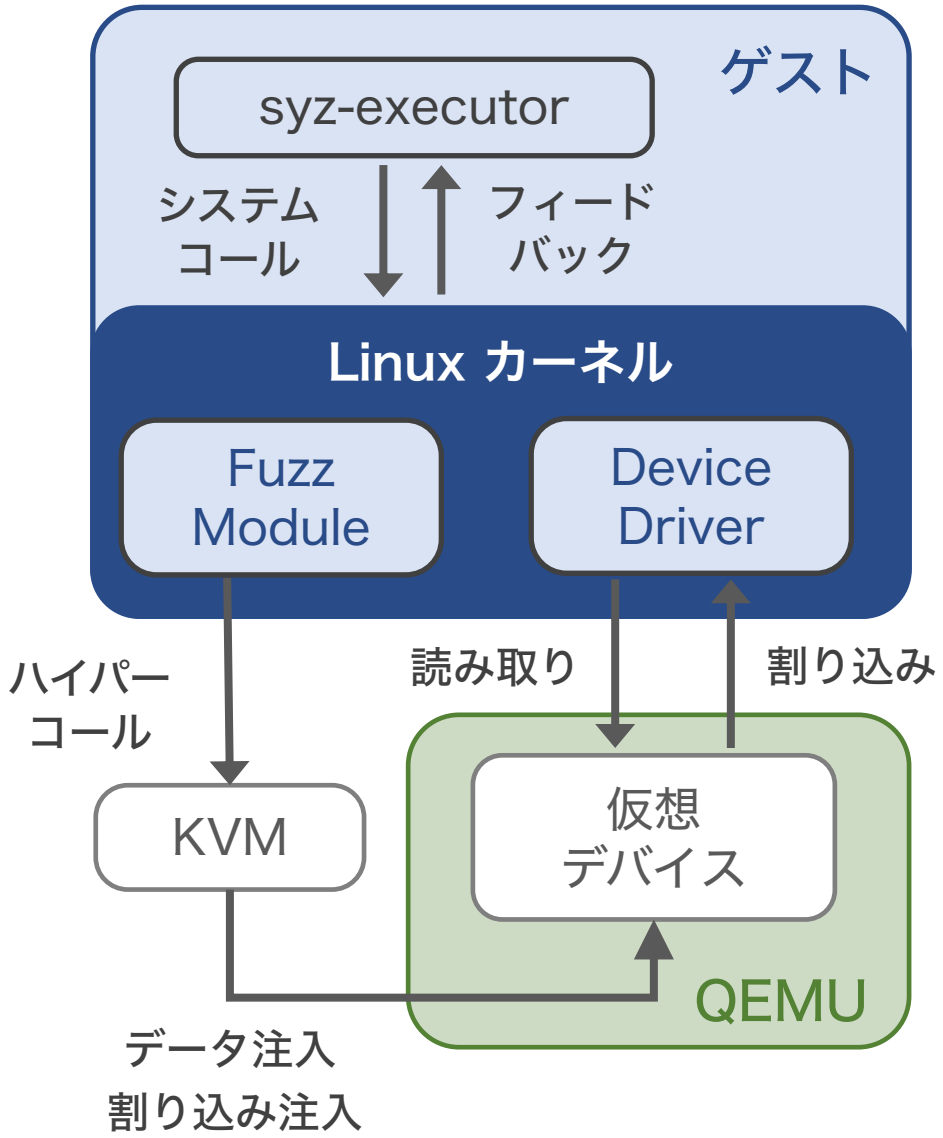
・ドライバはどのレジスタから読み込みを開始しても常にバッファの最初を返し，読み込みが完了するまでバッファポインタを次のバイトに移動する

→ メモリの削減



Multi-dimension Fuzzing

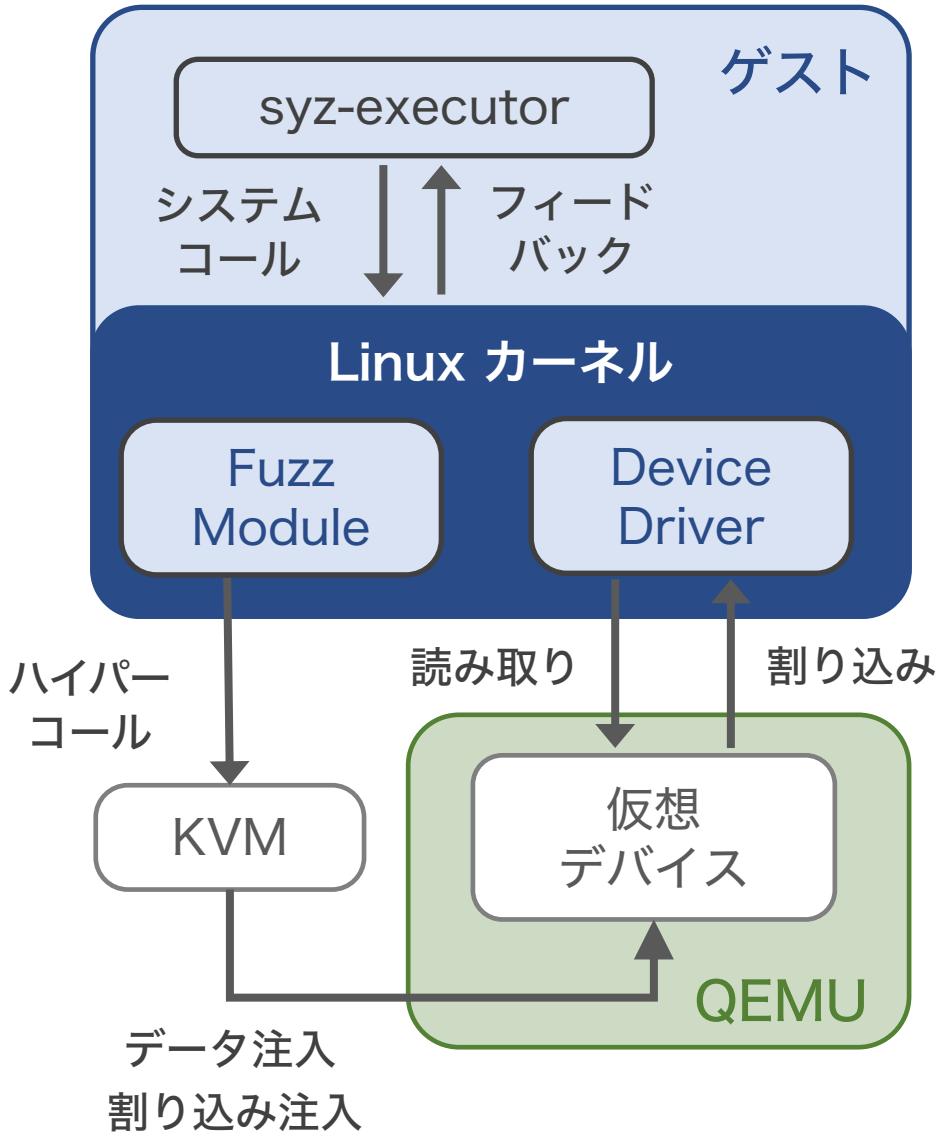
- PrIntFuzz は Syzkaller を使用して
  システムコールの列を生成
- 以下の2つのシステムコールを新規作成
  - データ注入システムコール
  - 割り込み注入システムコール
- 新たな2つのシステムコールは
  ハイパーコールを発生させ, KVM を通して
  QEMU にデータ / 割り込みを注入



Multi-dimension Fuzzing

・データ注入システムコールは
各通常システムコールの前に必ず
挿入されるように生成・変異の
アルゴリズムを変更

→ ドライバが各システムコール中に最新の
生成データを読み取ることができる



Implementation

- KCOV はソフトウェア割り込みからのカバレッジ収集のみをサポート

  KCOV [9] : Linux カーネルのコードカバレッジ収集フレームワーク

- PrIntFuzz ではハードウェア割り込みからのカバレッジ収集を
  サポートするように KCOV を修正

[9] https://lwn.net/Articles/671640/

Evaluation

- 5つの RQ
  - **RQ1**： PrIntFuzz はどの程度デバイスをシミュレートできるか？
  - **RQ2**： PrIntFuzz はどの程度バグを発見できるか？
  - **RQ3**： PrIntFuzz のコードカバレッジはどの程度か？
  - **RQ4**： PrIntFuzz はどの程度拡張性があるか？
  - **RQ5**： PrIntFuzz はどのようなバグを発見したか？
- 実験環境
  - Intel Core i9-12900KS
  - 128GB RAM
  - Ubuntu 20.04.4

Experiment Setup

- PrIntFuzz が特定したシステムコールのみでテスト
  → Syzkaller がサポートするすべてのシステムコールではない
- ドライバ間の干渉を避けるため, 一度に1つのドライバのみをサポート
- 各ドライバは一時間ファジングする
  → 1（時間）× 311（PCI device）× 4（種類）× 3（回）= 155（日）

RQ1：Virtual Device Modeling

- RQ1： PrIntFuzz はどの程度デバイスをシミュレートできるか？
  - Linux カーネル 5.18-rc1 上で 649 個の PCI バスに対応するデバイス ID を抽出
  - 311個（50.5%）のシミュレーションに成功
  - tty や ata ドライバの成功率が高い
    → 初期化時のチェックがあまりない or 簡単
  - sound や atm ドライバの成功率が低い

Type	Total	PrIntFuzz	Success Rate
net	164	71	43.3%
others*	147	62	42.2%
sound	58	17	29.3%
ata	56	53	94.6%
scsi	47	31	66.0%
media	34	16	47.1%
video	29	17	58.6%
i2c	17	9	52.9%
misc	16	9	56.3%
usb	14	8	57.1%
edac	12	5	41.7%
atm	11	3	27.3%
tty	11	10	90.9%
Total	616	311	50.5%

* All other types with less than 10 drivers.

RQ1：Virtual Device Modeling

・RQ1： PrIntFuzz はどの程度デバイスをシミュレートできるか？

シミュレーションに失敗する例

・レジスタが取り得る値の制約が複雑

・コンフィグレーションレジスタのアドレスが変数

・初期化中に特定の別のデバイスが存在しないとエラーを返す場合

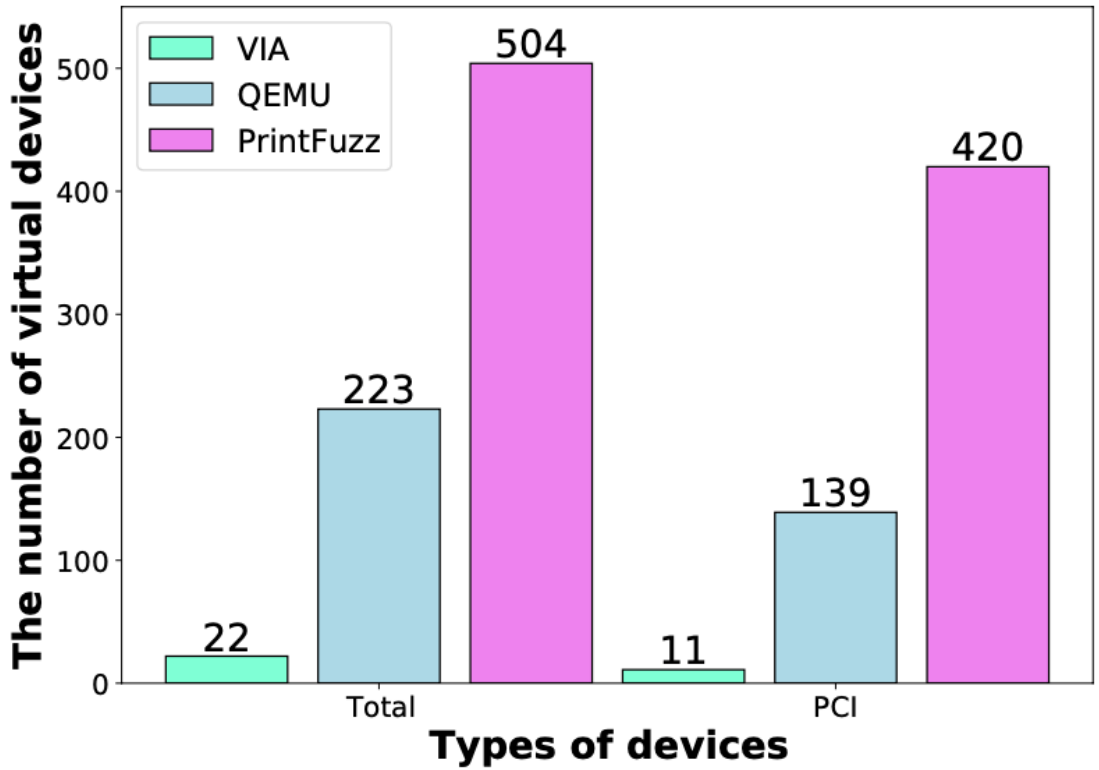
RQ1：Virtual Device Modeling

・RQ1： PrIntFuzz はどの程度デバイスをシミュレートできるか？

QEMU と VIA [5] との比較

・PrIntFuzz は QEMU よりも
281個（420 - 139）多くの
PCI デバイスをサポート

・VIA はより多くの仮想デバイスを
サポートするよう拡張できるが,
設定が手作業のため困難



[5] VIA: Analyzing Device Interfaces of Protected Virtual Machines [ACSAC'21]

RQ2 : Bug Finding in PCI Drivers

- RQ2 : PrIntFuzz はどの程度バグを発見できるか？
  - PrIntFuzz は PCI ドライバで
    他のファザーで発見されなかった
    112個のバグを発見
  - Null ポインタ参照は割り込みハンドラで多く発生する
    → 割り込みハンドラはデバイスから
    特定のレジスタを読み込んで
    状態を確認することが多く,
    無効なデータを与えると
    未初期化変数にアクセスすることも

Table 4: Bugs found by PrIntFuzz in PCI drivers

Type	Location			Number
	Main	Init	Interrupt	
Null-Pointer-Dereference	1	13	9	23
Divide-by-Zero	15	2	0	17
Use-After-Free	1	10	0	11
Page Fault	2	1	2	5
Out-of-Bounds	1	1	1	3
Wild-Memory-Access	1	0	0	1
Schedule-while-Atomic	0	0	1	1
Sleep-in-Atomic	0	1	0	1
Deadlock	1	0	0	1
Double Free	0	1	0	1
Warning	2	38	2	42
Assertion Failure	2	4	0	6
Total	26	71	15	112

RQ2 : Bug Finding in PCI Drivers

- RQ2 : PrIntFuzz はどの程度バグを発見できるか？
  - PrIntFuzz は PCI ドライバで
    他のファザーで発見されなかった
    112個のバグを発見
  - UAF バグは初期化やクリーン
    アップ機能でよく発生する
    → リソースの解放順の誤りや
    使用中のリソースを解放してしまう

Table 4: Bugs found by PrIntFuzz in PCI drivers

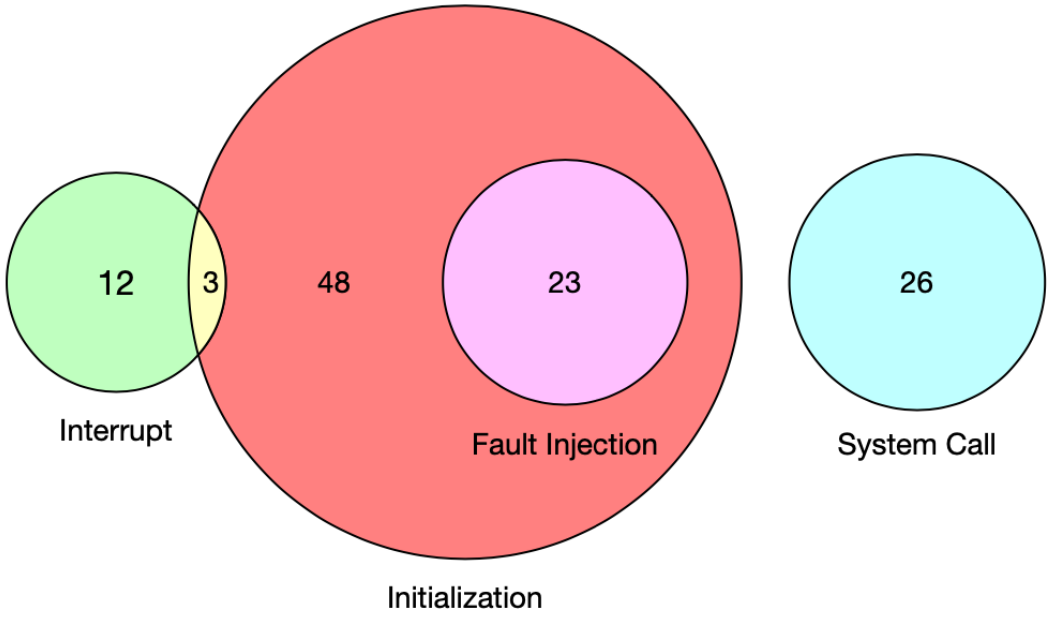
Type	Location			Number
	Main	Init	Interrupt	
Null-Pointer-Dereference	1	13	9	23
Divide-by-Zero	15	2	0	17
Use-After-Free	1	10	0	11
Page Fault	2	1	2	5
Out-of-Bounds	1	1	1	3
Wild-Memory-Access	1	0	0	1
Schedule-while-Atomic	0	0	1	1
Sleep-in-Atomic	0	1	0	1
Deadlock	1	0	0	1
Double Free	0	1	0	1
Warning	2	38	2	42
Assertion Failure	2	4	0	6
Total	26	71	15	112

RQ2 : Bug Finding in PCI Drivers

・RQ2 : PrIntFuzz はどの程度バグを発見できるか？

バグの分布

・初期化 + 割り込みで発生する
バグも存在する



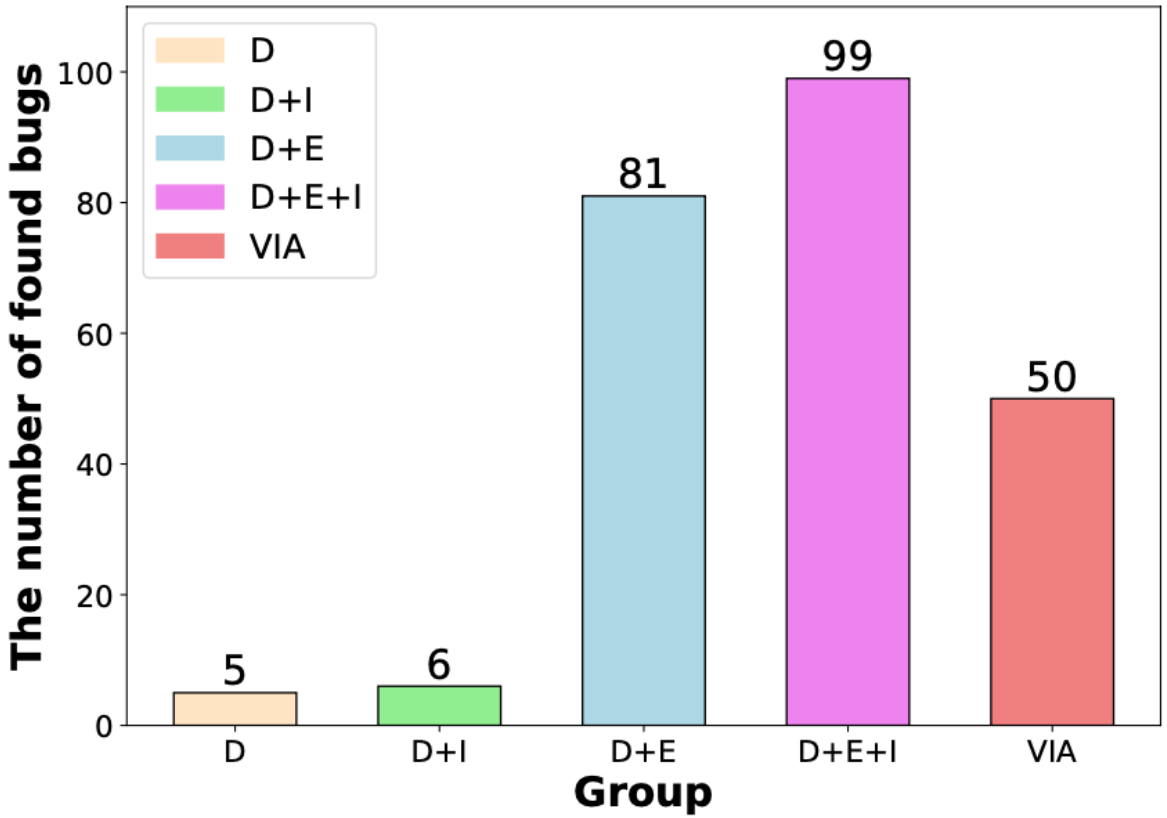
RQ2 : Bug Finding in PCI Drivers

・RQ2：PrIntFuzz はどの程度バグを発見できるか？

Syzkaller と VIA との比較

・3回の実験結果

No.	(D)Default Devices	(E)Extra Devices	(I)Interrupt
1	139	0	×
2	139	0	✓
3	139	281	×
4	139	281	✓



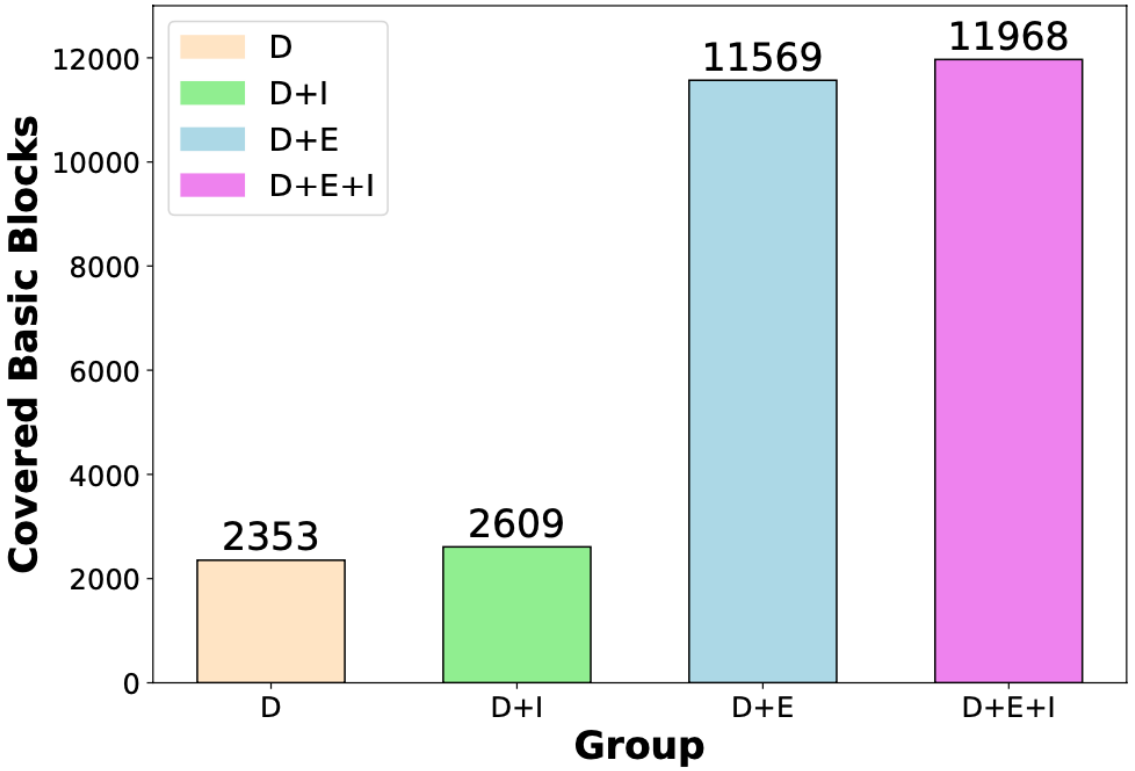
RQ3 : Code Coverage

- RQ3 : PrIntFuzz のコードカバレッジはどの程度か？

Syzkaller との比較

- 3回の実験結果の平均
- コアや共通処理のカバレッジは
  デフォルトの Syzkaller を実行して
  ベースラインとして設定し,
  そのカバレッジは除外

  → 初期化と割り込みハンドラの
  カバレッジの改善を考察したい



No.	(D)Default Devices	(E)Extra Devices	(I)Interrupt
1	139	0	×
2	139	0	✓
3	139	281	×
4	139	281	✓

RQ3 : Code Coverage

- RQ3 : PrIntFuzz のコードカバレッジはどの程度か？

  Agamotto [10] と VIA との比較

  - Agamotto と VIA がサポートする
    デバイスのみで比較
  - PrIntFuzz は gve ドライバをうまく
    初期化できない
  - nvme ドライバは対応する手動で
    書かれたハーネスが必要

Driver	Agamotto	VIA	PrIntFuzz
8139cp	4%	11%	26%
e100	3%	10%	47%
e1000	4%	16%	26%
e1000e	2%	9%	21%
gve	5%	18%	4%
ne2k-pci	1%	1%	29%
nvme	11%	25%	9%
rocker	1%	2%	3%
sungem	6%	8%	19%
sunhme	14%	18%	18%
vmxnet3	7%	13%	28%

[10] Agamotto: Accelerating kernel driver fuzzing with lightweight virtual machine checkpoints [SEC‘20]

RQ4 : Scalability

- RQ4 : PrIntFuzz はどの程度拡張性があるか？
  - PCI ドライバの例を見てきたが, I2C や USB にも同じ手法が適用できるか？
  - 仮想デバイスのモデリングは2つの側面に分けられる
    - MMIO, DMA, 割り込みなど本質的な機能のモデリング
      → インターフェースが同じなため, 直接移行できる
    - ドライバ固有の機能のモデリング
      → I2C はコンフィグレーションレジスタのモデリングが不必要
      識別子の文字列でデバイスを照合する
      → USB は readl のような関数を単純に使わずに
      URB (USB Request Block) を通して通信を行う
      → これらの機能も少しの修正で対応することができる

RQ4 : Scalability

- RQ4 : PrIntFuzz はどの程度拡張性があるか？
  - USB 169 / 346 (48.8%),
    I2C 472 / 895 (52.7%) の
    モデリングに成功
  - USB / I2C の初期化コードで
    18 / 20 個のバグを発見
    → USB / I2C では割り込みとデータ
    注入は実装されていない...

Type	USB	PrIntFuzz	I2C	PrIntFuzz
media	121	60	131	72
net	69	18	0	0
serial	49	34	0	0
others*	40	18	127	29
misc	22	14	16	12
input	19	13	0	0
storage	14	7	0	0
sound	12	5	100	64
iio	0	0	165	96
hwmon	0	0	146	110
input	0	0	73	7
regulator	0	0	43	27
mfd	0	0	38	12
rtc	0	0	30	27
power	0	0	26	16
Total	346	169	895	472
Success Rate	N/A	48.8%	N/A	52.7%

* All other types with less than 10 drivers.

RQ5 : Case Study

- RQ5 : PrIntFuzz はどのようなバグを発見したか？
  - `flexcop_pci_isr` 関数（割り込みハンドラ）における out-of-bounds
  - 原因は 1行目でレジスタの読み取り値をチェックしていないこと
  - 悪意のあるデバイスが大きな値を渡すと `buf` の領域外にアクセスする

```
1  dma_addr_t cur_addr = fc->read_ibi_reg(fc,dma1_008).dma_0x8.dma_cur_addr << 2;
2  u32 count = cur_addr - fc_pci->dma[0].dma_addr0;
3
4  while (pos < count) {
5      if (buf[pos] == 0x47)
6          break;
7      pos++;
8  }
```

Threats to Validity

- 仮想デバイスは MMIO, PIO, 割り込みなど基本的な機能のみ提供する
  - → より高度なセマンティック機能には対応できない
    - ex.) ネットワークの輻輳の処理
- フォルト注入における false-positive
  - → PrIntFuzz におけるフォルト注入は実際に関数を実行せずにエラーを返す
  - → しかしこのような誤検出は稀

```
1  if (foo(dev) < 0) {
2      put_device(dev->dev);
3      return ret;
4  }
```

```
1  int foo(struct foo_dev *dev) {
2      device_initialize(&dev->dev);
3      ...
4  }
```

関連研究

ファザー	対象OS	モデリング手法	MMIO	PIO	DMA	IRQ	IRQ Timing
USBFuzz [4]	Any	○ マニュアル	○	○	○	○	×
FuzzUSB [11]	Linux	○ マニュアル	×	×	×	×	×
DrFuzz [12]	Linux	○ 静的解析	○	○	×	×	×
PrIntFuzz [1]	Linux	○ 静的解析	○	○	○	○	×
Drifuzz [13]	Linux	○ コンコリック実行	○	○	○	○	×
DevFuzz [14]	Any	○ シンボリック	○	○	○	○	×
提案手法	Linux	× ー	○	○	○	○	○

関連研究

手法	対象（Linux）	並行性	UAF	割り込み
Syzkaller [15]	kernel	×	×	×
Actor [16]	kernel	×	○	×
Krace [17]	file system	○	×	×
DDRace [18]	driver	○	○	×
PrIntFuzz [1]	driver	×	×	○
DevFuzz [16]	driver	×	×	○
提案手法	driver	○	○	○

関連研究

[1] PrIntFuzz: fuzzing Linux drivers via automated virtual device simulation [ISSTA‘22]
[2] https://events.static.linuxfound.org/sites/events/files/slides/Android-%20protecting%20the%20kernel.pdf
[3] Understanding Linux kernel vulnerabilities. Journal of Computer Virology and Hacking Techniques (2021), 1-14
[4] USBFuzz: A framework for fuzzing USB drivers by device emulation [SEC’20]
[5] VIA: Analyzing Device Interfaces of Protected Virtual Machines [ACSAC‘21]
[6] https://www.usb.org/sites/default/files/vendor_ids100824_0.pdf
[7] Difuzz : Interface aware fuzzing for kernel drivers [CCS’17]
[8] https://github.com/ulfalizer/Kconfiglib/
[9] https://lwn.net/Articles/671640/
[10] Agamotto: Accelerating kernel driver fuzzing with lightweight virtual machine checkpoints [SEC‘20]
[11] FuzzUSB: Hybrid Stateful Fuzzing of USB Gadget Stacks
[12] Semantic-informeddriverfuzzing without both the hardware devices and the emulators [NDSS’22]
[13] Drifuzz: Harvesting bugs in device drivers from golden seeds
[14] DEVFUZZ: Automatic Device Model-Guided Device Driver Fuzzing [SP’23]
[15] https: //github.com/google/syzkaller
[16] ACTOR: Action-Guided Kernel Fuzzing [SEC’23]
[17] KRACE: Data Race Fuzzing for Kernel File Systems [SP’20]
[18] DDRace: Finding Concurrency UAF Vulnerabilities in Linux Drivers with Directed Fuzzing [SEC‘23]

まとめ

ジャンル： Linux デバイスドライバファジング

問題提起：
・既存のカーネルファザーはサポートするデバイスの **数・機能** が少ない
→ コードカバレッジの低下

提案手法：
・静的解析を利用した仮想デバイスの自動作成
・初期化処理へのソフトウェアフォルトの挿入
・システムコール・デバイス I/O ・割り込みといった複数の
入力ソースからの多次元ファジング

結果：
・311 / 472 / 169 個の PCI / I2C / USB デバイスを自動生成
・デバイスドライバの 150 個のバグを発見