カーネル内インタプリタに特化したファジングの提案

平松 勇人[1,a)]　石黒 健太[1,b)]　河野 健二[1,c)]

概要：Linux のカーネル内インタプリタである eBPF はセキュリティ機構として使われており，その脆弱性は致命的である．脆弱性を発見する手法としてファジングが利用されているものの，eBPF のコードサイズは Linux の 0.25%と小さく，Linux に対してファジングを行っても eBPF のコードが検査される比率は低い．また，eBPF インタプリタは入力の厳密な検査を行うため，ファジングにより生成された入力の多くは不正な入力として弾かれ，インタプリタ内部の脆弱性検査はできない．本論文では，eBPF を集中的にファジングするため，1)eBPF モジュールを通ったシードの優先度を高め，2) そうしたシードには細粒度のミューテンションを行うことで，eBPF を通る可能性の高いシードを効率的に生成する．また，eBPF の実行に必要な一連のシステムコールを束ねた擬似システムコールを含むシードを用意することで，不正な入力として弾かれることの少ないシードを生成する．Syzkaller に提案手法を実装し，eBPF に対してファジングを行った結果，従来の Syzkaller ではカバーできなかった 314 箇所のコードをカバーできることを確認した．

A fuzzing method specialized in the in-kernel interpreter

Hayato Hiramatsu[1,a)]　Kenta Ishiguro[1,b)]　Kenji Kono[1,c)]

Abstract: eBPF is an in-kernel interpreter of Linux kernel and used as an implementation platform to enhance security. Thus, vulnerabilities in eBPF are critical. Fuzzing is proven to be useful to find vulnerabilities and widely used. However, the size of eBPF module is only 0.25% of Linux and it is difficult to fuzz eBPF while fuzzing Linux. Because eBPF verifies its inputs, most of them are rejected as malformed and fuzzers cannot fuzz eBPF's deep internals. In this paper, we propose a fuzzing method to efficiently test eBPF. First, we prioritize seeds which exercised eBPF. Second, we mutate them in more fine-grained ways to efficiently generate seeds which more likely to exercise eBPF. And we create pseudo syscalls which include multiple syscalls, to generate seeds which are less likely to be rejected by the verifier. We implemented it on Syzkaller and we found that our method has covered 314 more codes than Syzkaller.

1. はじめに

Linux カーネルは現在幅広い分野で使用されており，そのセキュリティは重要となっている．しかし，Linux カーネルは多数のコントリビュータによって開発されており，2000 万行以上のソースコードで書かれているため，バグがほとんどないとはいえない状況にある [1]．カーネルモジュールの中でも eBPF は，システムコールのフィルタリング，パケットフィルタ，プログラムのトレースとプロファイリング，カーネルのモニタリングなどに使用することができる Linux のカーネル内インタプリタである．セキュリティ機構として使用されている [2] ため，その脆弱性は致命的となる．

ソフトウェアのテスト手法のひとつであるファジングは，テストケースを自動的に生成するテストとして実用的に使われている [3][4]．フィードバックとしてコードカバレッジを利用するカバレッジガイデッドファジングでは，

(1) シードを生成する
(2) 実際に実行を行う
(3) 実行の結果としてコードカバレッジを得る
(4) コードカバレッジを用いてシードの評価を行う

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

(5) シードを保存する
(6) シードをミューテーションし，新たなシードを作る
(7) 再び実行する

というループを行う．この過程で，シードはより多くのコードカバレッジを達成するようになり，より多くのバグを発見する可能性が高まる．

オペレーティングシステムに対してもファジングは用いられており，システムコールをインターフェースとするシステムコールファザーが主流となっている [5][6][7]．システムコールファザーではシードはシステムコールの列であるプログラムとなる．Linux カーネルは kcov という仕組みによりカーネルのコードカバレッジの情報を取得することができるため，カバレッジガイデッドファジングが可能になっている．しかし，Linux カーネルに対し eBPF モジュールのコードサイズは 0.25% と小さく，ファジングで eBPF モジュールを実行するシードが自然に生成されるのを待つことは非常に非効率的となる．また，eBPF はベリファイアで不正な入力を弾く機構を持つため，ファジングで入力を正しく生成することが難しくファジングが進まない要因となっている．

ソフトウェアの一部分に集中的にファジングを行う，ダイレクテッドファジングが提案されている [8][9]．ここではアプリケーション全体を静的解析し，ターゲットとなるコード領域により大きな重みを与えることによって，ターゲットにより近いシードにより高い評価を与える．ターゲットを実行するシードの実行回数とミューテーション回数を増加させる．しかし，ユーザ空間のアプリケーションに対して用いられている手法を Linux カーネル全体に対し適用しそれぞれのアドレスに対し重みを変化させることはスケールしないと考えられる．Hawkeye[9] では，260 行の C で書かれた Binutils の cxxfilt を静的解析するために 12.5 分かかっている．解析時間がコードサイズに比例するとしても Linux では 2 年近くかかる．

本論文では，3 つのアプローチを用いて，eBPF モジュールを集中的にファジングする手法を提案する．まず，eBPF モジュールを通ったシードの優先度を増加させる優先度ブーストを行い，優先的にスケジューリングを行うようにする．一度実行したコードカバレッジを元に優先度を増加させるため，Linux カーネル全体を静的解析する必要がないためスケールする．2 つ目に，優先度がブーストされたシードに対して粒度の小さいミューテーションを行う，アダプティブミューテーションを行うようにする．優先度が高いシードはターゲットに近いと考えることができるため，実行パスを大きく変更しないようなミューテーションを適用する．これにより，再び eBPF モジュールを通る可能性を上昇させる．3 つ目として，eBPF のシステムコールを実行するために必要な条件を満たすように，一連のシステムコールを束ねた疑似システムコールを用意し，シードとして利用できるようにする．これにより不正な入力としてベリファイアに弾かれることを少なくすることができる．Syzkaller に提案手法を実装し，eBPF に対してファジングを行った結果，従来の Syzkaller ではカバーできなかった 314 箇所のコードをカバーできることを確認した．

```
1 BPF_ST_MEM(BPF_DW, BPF_REG_10, −8, 0),
2 BPF_MOV64_REG(BPF_REG_2, BPF_REG_10),
3 BPF_ALU64_IMM(BPF_ADD, BPF_REG_2, 0),
```

図 1 eBPF プログラム例

2. 背景

2.1 Linux カーネルの重要性

Linux カーネルは現在広く使われており，そのセキュリティは重要となっている．2004 年から 2010 年までにリリースされた Linux の各バージョンについて，きわめて単純な静的解析で見つけることのできるバグを調査したところ，各バージョンに常に 600 個程度のバグが存在していたことが報告されている [1]．ここで見つけているバグは，例えば `malloc` の戻り値をキャストした型と，`malloc` で確保したデータサイズが一致しないなど，ポインタ解析など複雑な解析を一切必要としないものばかりである．そのため，Linux カーネルでは頻繁にコード修正が行われており，そのようなコード修正が新たな脆弱性を導入していないかどうか検査する必要がある．ユーザ空間のアプリケーションのパッチに関する研究では，リリース後のパッチのうち 18% が不完全なものだったという報告がされている [10]．さらにこのうち 43% は新たに深刻なバグを発生させていた．そこで，Linux のような大規模なソフトウェアであっても，修正を行なったモジュール単位で効率的にパッチテストを行うことが望ましい．しかし，カーネルのモジュールは相互にデータ構造を共有しながら動作していることが多く，特定のモジュールのみを切り出してパッチテストを行うことは難しい．本論文では eBPF モジュールを対象に効果的にテストを行う手法を検討する．eBPF モジュールは，Linux カーネルのカーネル内インタプリタである．システムコールのフィルタリング，パケットの処理の記述を行うことでセキュリティを向上させることができる．さらに，プログラムのトレースとプロファイリング，カーネルのモニタリングなどに使用することができる [11]．また，セキュリティのフレームワークを実装する基盤として使用されている [2]．このため，eBPF をテストすることが重要となっている．

2.2 eBPF プログラム

図 1 に eBPF のプログラム例を示す．行 1 はスタックのメモリの初期化を行う．行 2 はレジスタが初期化したメ

モリを指すようにする．行 3 はレジスタを通じてスタックに値を代入する．このあとに eBPF のヘルパ関数を呼び出すことで様々な機能が利用できる．

行 1 がない場合はメモリの初期化がされていないためエラーとなる．このように eBPF の命令間には依存関係が存在するためテストケースを自動生成する場合はベリファイアに弾かれないように正しい入力を作らなければならない．

2.3 カバレッジガイデッドファジング

ファジングはテストケースを自動的に生成し実行を行うテスト手法のひとつで，実際にアプリケーションのテストに用いられている [3][4]．

カバレッジガイデッドファジングでは，より多くのコードをカバーすればより多くのバグを発見する確率が高まるという考えを元に，実行結果のコードカバレッジを利用してファジングプロセスを効率化している．

ファジング開始前に，コードカバレッジを得られるようにインストルメントしたバイナリを用意しておき，ファジング中の動作は以下のようになる．1) シードを生成する．2) 実際に実行を行う．3) 実行の結果としてコードカバレッジを得る．4) コードカバレッジを用いてシードの評価を行う．5) シードを保存する．6) シードをミューテーションし，新たなシードを作る．評価値が高かったシードは何度もミューテーションを行う．7) 再び実行する．

シードの評価の観点は 1) 新たなコードカバレッジを示したか，2) 正常に終了したか，などがある．

2.4 ダイレクテッドファジング

アプリケーションの特定のコード領域に集中的にファジングを行うファジングはダイレクテッドファジングと呼ばれている．ユーザ空間のアプリケーションに対する研究がなされている [8][9]．

パッチテスティングとリグレッションテスト，詳細な情報を含めることができないクラッシュレポートの再現，静的解析の結果を利用しバグの可能性のあるコードを確認するといったことに利用することができる [8]．

ファジング開始前に，アプリケーションを静的解析しベーシックブロックや関数ごとに重みを与える．ファジング中の動作は 2.3 章のカバレッジガイデッドファジングと同じだが，シードの評価をする段階では重みを考慮してカバレッジからシードの評価を行う．評価値が高いシードはよりターゲットに近い，逆に評価値が低いシードはターゲットから遠いとして，評価値がターゲットとの距離を表すようにする．

Böhme ら [8] はシードの評価値を出すため，ターゲットとなるベーシックブロックとのコールグラフとコントロールフローグラフでの距離を用いている．シードの評価値をシードの試行回数として用いることで，より高優先度，つまりターゲットに近いシードの試行回数を増加させダイレクテッドファジングを実現している．このスケジューリングがパワースケジューリングである．

Chen ら [9] はダイレクテッドファジングでのミューテーションにおいてターゲットとの距離であるシードの評価値を使いミューテーションの内容を変え，よりターゲットを実行するようなシードを作っている．ミューテーションの内容は以下のように分類し適用している．1) ターゲットに近いシードは粒度の小さいミューテーションを行う．例: ビットの反転や算術演算．2) ターゲットから遠いシードは粒度の大きいミューテーションを行う．例: 複数バイトの書き換え，抽象構文木レベルでの変換，他シードとの交差．

2.5 カーネルファジング

カーネルに対するファジングでは，システムコールのインターフェースを利用するシステムコールファジングが広く利用されている [5][6][7]．Syzkaller[5] は実際に 500 以上のバグを発見している．Linux カーネルでコードカバレッジを得るために使用することができる kcov はシステムコールの入力に対して特化している [12]．システムコールファジングでは，シードはシステムコール列であるプログラムとなり，そのプログラムをコンパイルして実行することになる．

カーネルファジングではミューテーションは以下の 3 種類となる．1) システムコールの引数の変更，2) システムコールの挿入・削除，3) 他シードとの交差．シードの評価の観点は以下のようになる．1) 新たなコードカバレッジを示したか，2) システムコールが正常に終了したか．

システムコールは引数に構造体を要求するなど，複雑で正確な入力を必要とする．Syzkaller[5] では，それぞれのシステムコールの引数の取りうる範囲を形式的に記述したテキストファイルを前もって作成することで，引数として正しいシステムコール列を生成している．また，システムコールの戻り値を他のシステムコールの引数とすることも可能にしている．

3. 問題

eBPF モジュールに対するダイレクテッドファジングを実現する上で 3 つの問題がある．

3.1 静的解析がスケールしない

ユーザ空間のアプリケーションに対して用いられてきた静的解析の結果を重みとして利用する手法は，カーネルはコードが巨大であるためスケールしないと考えられる．また，カーネル内の様々な変数やファイルシステムの状態などが関わってくるため，静的解析でターゲットとの距離を算出することが難しくなっている．

3.2 他のモジュールに入ってしまう

カーネルのコードサイズはとても大きい．このため偶然最初に実行したカーネルのコンポーネントから大きなカバレッジを得ることになる．カバレッジガイデッドファザーはそのシードを何度もミューテーションし実行するので他のシードの試行回数が減少してしまう．eBPF を実行するシードを優先する仕組みが必要である．

3.3 プログラムのミューテーション

eBPF モジュールを実行するシードに対してミューテーションを行うとき，ミューテーション後の新たなシードは eBPF モジュール内を探索するようなシードになるようにしなければならない．

3.4 eBPF のベリファイア

eBPF は入力となる eBPF プログラムを厳しくチェックする．このため，eBPF システムコールの引数の型と値の範囲が正しくなるように生成しても，eBPF プログラムの命令間の依存関係は全く無視しているため正しい eBPF プログラムとならず，不正な入力として弾かれてしまう．ベリファイアに弾かれないために複数のシステムコールを正しく組み合わせて使用する必要がある．

4. アプローチ

本論文では，eBPF を集中的にファジングするため，3 つのアプローチを提案する．

4.1 パワースケジューリング

ダイレクテッドファジングにおけるパワースケジューリングとは，コーパス内でシードを選び出し実行する回数をシードの評価に応じて変更することである [13]．この実行する回数は，

- コーパスからシードを選び出し繰り返し実行する回数
- ミューテーション時，元として使用するシードとして選び出す回数

の二つに分けて考えることができる．前者はグローバル変数などのカーネルの内部状態が変化した場合のため再び同じ検査をすることになる．後者はターゲットモジュールをファジングするようなシードを生成する頻度を上げることができる．

本論文ではこれらの回数をシードの優先度を増加させることで操作する．シードの優先度は，一度シードを実行した結果，ターゲットとなるモジュールのカバレッジに比例した値とする．ターゲットのアドレス範囲を前もって算出しておき，出力されたコードカバレッジがその範囲内であった場合優先度を増加させる．ループや，シード内に繰り返し同じシステムコールが含まれていた場合などにより，ターゲット内の同じアドレスを何度も通った場合は，その回数に比例して優先度を増加させる．

この仕組みによりファジング中に新たにターゲットのカバレッジがより高いシードが見つかった場合，そのシードを最も高い優先度とすることができる．

あるシード x を選び出す確率 $P(x)$ はシード x の優先度を $Prio(x)$ とすると

$$P(x) = \frac{Prio(x)}{\Sigma_k Prio(k)} \tag{1}$$

とする．ここで分母は x を含む全てのシードの優先度の総和である．

式 1 により，ファジング中に新たにより良いシードが見つかった場合，そのシードが選ばれる確率を最も高くすることができる．また，シードの選出は一定時間で行うことができるため，ファジングを遅らせることはない．

4.2 アダプティブミューテーション

あるシードが eBPF モジュールを実行していると分かった場合，シードを大きくミューテーションしてしまうと，再び eBPF のベリファイアに不正な入力として弾かれてしまう可能性が高くなってしまうと考えられる．逆に，あるシードが eBPF モジュールと関係のないモジュールをファジングしている場合，シードを大きくミューテーションし，カーネル内を広く探索しながら eBPF モジュールへのエントリーポイントを探すことが必要である．

アダプティブミューテーションでは，ターゲットに近いと考えられるシードのミューテーションをより小さく，ターゲットから遠いと考えられるシードのミューテーションを大きく行う．

ミューテーションの手法と実行パスの関係性については，ユーザ空間のアプリケーションに対するファジングにおいてシードの交差が実行パスの変化が大きいと分かっている [14]．これをシステムコールファザーで用いられているミューテーション手法に対して適用すると，最も実行パスの変化が小さい順に，

(1) システムコールの引数の変更
(2) システムコールの挿入・削除
(3) 他シードとの交差

と考えられる．

ここで (3) は複数のシステムコールの挿入を行うミューテーションの一種だと言うことができる．このため，(2) は実行パスの変化が中程度であるが，繰り返しミューテーションを行うことで実行パスの変化が大きくなると考えることができる．

アダプティブミューテーションでは，パワースケジューリングで用いたシードの優先度を用いる．優先度が低い場合，他のシードとの交叉も含めミューテーションを行う．優先度が高い場合，ミューテーションの回数を減少させるとともに，ミューテーションの内容も実行パスの変化が小

さいミューテーションを選ぶ確率を上げる．

4.3 疑似システムコール

カーネル内インタプリタである eBPF モジュールを実行するために bpf() システムコールがある．第一引数にコマンドとして整数値を取り，それにより様々な eBPF に関する操作をすることができる．

bpf() システムコールが eBPF モジュールのエントリーポイントとして使用できるため，システムコールファザーが eBPF モジュールをファジングすることは一見容易に思えるが，コマンドの中には複雑な入力を必要とするものがある．BPF_PROG_LOAD コマンドはその一つであり，eBPF プログラムを入力として受け取り，ベリファイし，ロードしたあとファイルディスクリプタを返戻するコマンドである．

この BPF_PROG_LOAD コマンドでは入力のチェックが厳しく行われる．eBPF プログラムを入力した場合，

- 有向非巡回グラフを用いてループなどのチェックを行う．ここで，到達不可能な命令が存在しないか検知する．
- 最初の命令から順に全ての実行パスをシミュレーションし，レジスタとスタックのチェックを行う．初期化が行われているか，型が正しいかを見る．

というチェックがベリファイアで行われる．

このチェックの過程で不正な入力として弾かれる例を列挙すると以下のようになる．

- 到達不可能な命令がある
- レジスタが初期化されていない
- スタックが初期化されていない
- スタックの境界外アクセスをしている
- ヘルパ関数を呼び出したとき戻り値をチェックしていない
- eBPF の map へのファイルディスクリプタであるはずの引数が map ではない
- map のキーやバリューの型が違う

疑似システムコールとして一連のシステムコールを束ねて実行するようにすることで，eBPF プログラム内での命令同士の依存関係を正しく生成できるようにする．また，他の bpf() システムコールでデータ構造を前もって作るようにする．これにより不正な入力として弾かれにくくする．実装する上で，それぞれについて見ていく．

4.3.1 到達不可能な命令の排除

exit 命令や jmp 命令により到達不可能な命令が発生する．今回は exit 命令はプログラムの最後に一つのみとし，無条件の jmp は行わないこととした．

4.3.2 レジスタの初期化

レジスタは後に値が代入される場合であっても，型の整合性チェックのため初期化する必要がある．値の代入とその直前の初期化をひとまとめに扱うことで常に初期化することができる．

4.3.3 スタックの初期化，境界外アクセスの排除

スタックについてもレジスタと同様に，スタックポインタを増加させたあと初期化が必要である．これらをひとまとめに扱う．

4.3.4 Map への適切なアクセス

eBPF のデータ構造の一つである map は bpf() システムコールの BPF_MAP_CREATE コマンドで作成することができる．作成時に，map のタイプをハッシュテーブル，配列などから指定することができ，map のキーサイズ，バリューサイズ，最大要素数を指定する．

map を作成した後にファイルディスクリプタを引数にとる命令に渡すようにし，そのあとの命令ではキーとバリューのサイズを一致させるように使用することで map への適切なアクセスが可能になる．

5. 実験

提案手法のプロトタイプを Syzkaller[5] に対して実装した．Syzkaller は現在最も活発に開発が行われているシステムコールファザーのひとつである．疑似システムコールに関して，特に自由度を下げて実装を行い，ベリファイアに弾かれないことを優先して実装を行った．

全体のコードカバレッジ，ターゲットである eBPF モジュールのカバレッジの増減について Syzkaller と比較を行った．eBPF モジュール内で新たにカバーできたコードのカバーできるようになった理由を調べ，提案手法が有効であることを確かめる．

5.1 実験環境

実験環境は，Intel Xeon E-2226G CPU 3.40GHz を使用し，ホスト OS は Debian 10.8 である．メモリは 64GB である．ファジング対象のゲスト OS として Linux Kernel 5.6.0-rc4 を用いた．

初期コーパスとして eBPF を実行するシード群を含むコーパスを使用し，ターゲットを eBPF 関連のファイル群である，net/core/filter.c，arch/x86/net/bpf_jit_comp.c と kernel/bpf 以下の全ファイルとした．提案手法，Syzkaller それぞれを 8 時間実行した．

5.2 コードカバレッジの増加

Linux カーネル全体のコードカバレッジが大きく減少してしまった場合，カーネル内を広く探索することができずにファジングとして様々な入力が作り出せていないと考えることができる．

図 2 に Linux カーネル全体のコードカバレッジを示す．

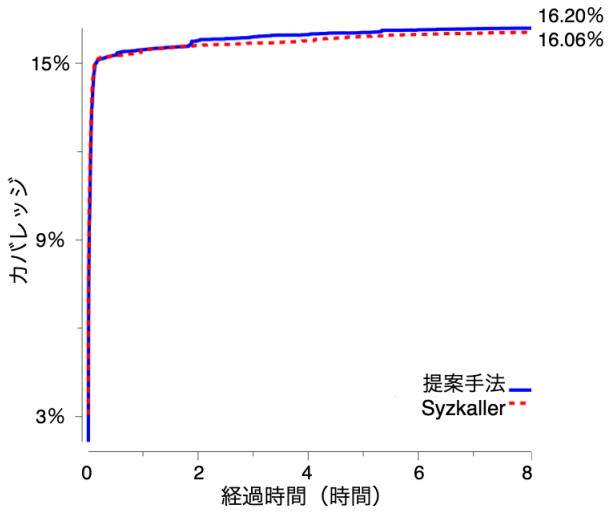

図 2 Linux カーネル全体のコードカバレッジ

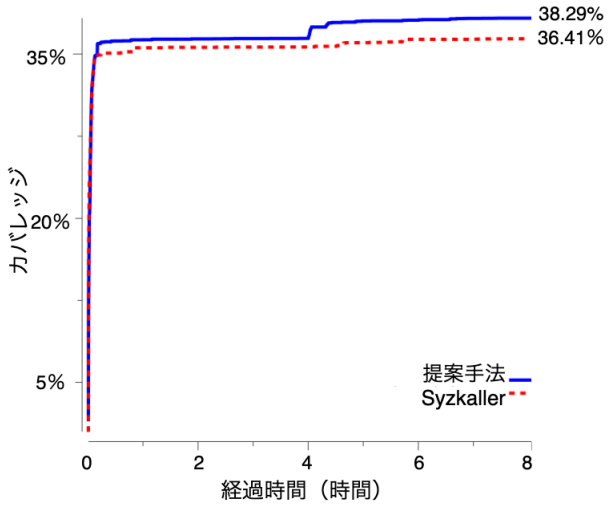

図 3 eBPF モジュールのコードカバレッジ

提案手法の場合は 16.20% となり，Syzkaller の 16.06% と比較してコードカバレッジは微増していることが分かる．このため，カーネル内を広く探索することは提案手法でもできていると言える．

5.3 ターゲットのカバレッジの増加

図 3 に eBPF ターゲットのコードカバレッジを示す．

図 3 では，ターゲットである eBPF モジュールのコードカバレッジについて，提案手法は 38.29% となり，Syzkaller は 36.41%となった．提案手法が 1.88% 高いコードカバレッジを達成していることが分かる．これはコードにすると 314 箇所となる．

ダイレクテッドファジングにより提案手法では，カーネル内の他のコンポーネントをファジングし続けてしまうということを防ぐことができていると分かる．

表 1 はターゲットごとのカバレッジの増減を示す．ターゲットとなったそれぞれに対しカバレッジが増加していることがわかる．`kernel/bpf` 内のそれぞれのファイルに対しては提案手法がより小さいカバレッジを示したファイルがあるが全体として増加となった．

表 1 ターゲットのファイルごとのカバレッジの違い

ファイル名	Syzkaller	提案手法	差
net/core/filter.c	13%	18%	+5%
arch/x86/net/bpf_jit_comp.c	44%	45%	+1%
kernel/bpf 以下の全ファイル	42%	43%	+1%

```
1  static int check_mem_access(..)
2  {
3      /* フラグが正しいかのチェック */
4      size = bpf_size_to_bytes(..);
5      /* ポインタのアラインメントのチェック */
6      err = check_ptr_alignment(..);
7
8      /* レジスタの型によって異なるチェックを行う */
9      if (reg->type == PTR_TO_MAP_VALUE) {
10         /* マップ型の場合 */
11     } else if (reg->type == PTR_TO_STACK) {
12         /* スタックの場合 */
13     } else {
14         /* エラーで返る */
15     }
16 }
```

図 4 簡略化したメモリアクセスチェックの関数 kernel/bpf/verifier.c

5.4 新たにカバーできたコードの例

5.4.1 メモリアクセスチェックのコード

図 4 はメモリアクセスのチェックをディスパッチする関数である．ここでまず最も基本的なメモリのチェックが行われ，次にそれぞれのレジスタの型に応じてチェックを行う．

行 4,6 ではフラグが正しいこととポインタのアラインメントをチェックする．行 4 はメモリアクセスを行うオペコードの 4 ビット目または 5 ビット目が 1 であることを見ている．オペコードは 8 ビットであるためランダムに生成したオペコードでこのチェックを通る確率は 1/4 となる．疑似システムコールによりこれらのチェックは通りやすくなっている．

行 10,12 のコードが提案手法により新たにカバーされた．ベリファイアは eBPF プログラムを一行ずつ見てレジスタの型を推論しているためこのコードのカバーは難しい．疑似システムコールによってレジスタの型が正しくなるようなプログラムを生成できたと分かる．

5.4.2 スタック境界のチェック

図 5 はスタックの境界チェック周辺のコードを形式的に表している．eBPF プログラムでヘルパ関数を呼び出している場合にこのコードが実行され，境界チェックが行わ

```
1 if (opcode == BPF_CALL) {
2     /* 呼び出すヘルパ関数のプロトタイプのチェック */
3     for (/* ヘルパ関数の引数それぞれについて */) {
4         /* スタックの境界チェック */
5     }
6 }
```

図 5 スタックの境界チェックコード kernel/bpf/verifier.c

れる．この `if` 節全体が Syzkaller ではカバーできていなかったが，提案手法により一部をカバーすることができた．

まず関数呼び出しのオペコードが正しく，設定されている必要がある．関数のプロトタイプから eBPF プログラムで使用できる関数か調べ，できない場合はエラーとなりそれ以降のチェックコードが一切カバーできない．次にヘルパ関数の引数のそれぞれについてスタックの境界チェックを行う．

このようなチェックがあるためヘルパ関数の呼び出しを行う eBPF プログラムをファジングで自然に作り出すことが難しい．Syzkaller では eBPF プログラムを生成する場合，システムコールの引数とシステムコール間の依存関係は正しく生成することができるが，eBPF プログラム内の依存関係を考慮することができていない．依存関係を正しく作る疑似システムコールによりカバーができたと考えられる．

6. 関連研究

6.1 ダイレクテッドファジング

Directed Greybox Fuzzing[8] はダイレクテッドファジングを提案した．ターゲットとなるベーシックブロックとの距離をコールグラフとコントロールフローグラフを用いて表現し，シードの評価値は焼きなまし法を用いて時間経過に応じて低下するようにして局所最適解に陥らない工夫をしている．

Muzz[15] はマルチスレッドのプログラムに対しインストルメントを最適化しスレッドのインターリーブを考慮したカバレッジを得ることで，効率的なマルチスレッドのプログラムのファジングを実現している．コード領域ではなくバグの種類に対して集中的にファジングを行うダイレクテッドファザーである．Krace[16] はカーネルのファイルシステムに対し，データレースを引き起こすような入力を作る．ファイルシステムのデータレースバグを引き起こすコードへのダイレクテッドファジングである．ParmeSan[17] はサニタイザーによりターゲットを決めダイレクテッドファジングを行う．

本論文では，カーネルに対し適用するために正確さよりスケールすることを求め，実際に実行したカバレッジを利用してターゲットとの距離を算出している．

6.2 カーネルファジング

Razzer[18] は，静的解析を行いデータレースの可能性のあるコードを発見し，ハイパーバイザを利用してスレッドのインターリーブを決定的に操作するカーネルファザーである．HFL[19] は，シンボリックエグゼキューションを利用し厳しい制約を通るような入力を作る，ハイブリッドファザーかつシステムコールファザーである．Moonshine[20] は実際のアプリケーションの実行のトレースからシステムコール間の依存関係を抽出し，システムコール列を生成するシステムコールファザーである．

本論文では，疑似システムコールとして複数のシステムコールを束ねて利用することでシステムコール間の依存関係を一部固定化し，eBPF プログラム内の命令間での依存関係を指定したと言うことができる．

6.3 プログラミング言語処理系に対するファジング

Favocado[21] は，Javascript のバインディングレイヤーに集中しファジングを行う．API リファレンスから意味論的情報を抽出し，構文的，意味論的に正しい入力を作り出す．ミューテーションはランダムに行う．P4Fuzz[22] は，プログラマブルなデータプレーンを記述するドメイン固有言語である P4 に対しファジングを行う，ミューテーションを行わない，ジェネレーションベースのファザーである．

本論文は eBPF に対し，システムコール列をミューテーションするミューテーショナルファザーである．現在の実装では疑似システムコールはジェネレーションベースに近く，ミューテーションのための十分なフィードバックを得ることができていないという制限がある．

7. おわりに

Linux カーネルは様々な場面で使用されておりそのセキュリティは重要となっている．カーネル内インタプリタである eBPF モジュールはセキュリティ機構として使われており重要であり，集中的にテストを行いたいという要求がある．カーネルに対するファジングが研究されているが，カーネルのコードは大きいため，従来のダイレクテッドファジングは容易に適用できない．さらに，eBPF モジュールは入力をチェックし不正な入力を弾くためファジングで入力を作ることが難しい．本論文では eBPF モジュールに対し集中的にファジングを行う手法を提案した．実行後のカバレッジを利用してシードの優先度を計算し，パワースケジューリングとアダプティブミューテーションを行う．疑似システムコールにより複数のシステムコールを束ね依存関係が正しい入力を生成する．

今後は，パワースケジューリングとアダプティブミューテーションのパラメータの調整と評価を行う．

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] Palix, N., Thomas, G., Saha, S., Calvès, C., Lawall, J. and Muller, G.: Faults in Linux: Ten Years Later, *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, New York, NY, USA, Association for Computing Machinery, p. 305–318 (online), DOI: 10.1145/1950365.1950401 (2011).

[2] Tian, D. J., Hernandez, G., Choi, J. I., Frost, V., Johnson, P. C. and Butler, K. R. B.: LBM: A Security Framework for Peripherals within the Linux Kernel, *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 967–984 (online), DOI: 10.1109/SP.2019.00041 (2019).

[3] Zalewski, M.: american fuzzy lop (2.52b), `https://lcamtuf.coredump.cx/afl/`.

[4] Google: OSS-Fuzz, `https://github.com/google/oss-fuzz`.

[5] Google: Syzkaller, `https://github.com/google/syzkaller`.

[6] Jones, D.: Trinity: Linux system call fuzzer, `https://github.com/kernelslacker/trinity`.

[7] Schumilo, S., Aschermann, C., Gawlik, R., Schinzel, S. and Holz, T.: kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels, *26th USENIX Security Symposium (USENIX Security 17)*, Vancouver, BC, USENIX Association, pp. 167–182 (online), available from ⟨`https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schumilo`⟩ (2017).

[8] Böhme, M., Pham, V.-T., Nguyen, M.-D. and Roychoudhury, A.: Directed Greybox Fuzzing, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, New York, NY, USA, Association for Computing Machinery, p. 2329–2344 (online), DOI: 10.1145/3133956.3134020 (2017).

[9] Chen, H., Xue, Y., Li, Y., Chen, B., Xie, X., Wu, X. and Liu, Y.: Hawkeye: Towards a Desired Directed Grey-Box Fuzzer, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, New York, NY, USA, Association for Computing Machinery, p. 2095–2108 (online), DOI: 10.1145/3243734.3243849 (2018).

[10] Yin, Z., Yuan, D., Zhou, Y., Pasupathy, S. and Bairavasundaram, L.: How Do Fixes Become Bugs?, *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, New York, NY, USA, Association for Computing Machinery, p. 26–36 (online), DOI: 10.1145/2025113.2025121 (2011).

[11] Cilium: What is eBPF? An Introduction and Deep Dive into the eBPF Technology, `https://ebpf.io/what-is-ebpf/`.

[12] kernel development community, T.: kcov: code coverage for fuzzing, `https://www.kernel.org/doc/html/v5.6/dev-tools/kcov.html`.

[13] Böhme, M., Pham, V.-T. and Roychoudhury, A.: Coverage-Based Greybox Fuzzing as Markov Chain, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, New York, NY, USA, Association for Computing Machinery, p. 1032–1043 (online), DOI: 10.1145/2976749.2978428 (2016).

[14] Blair, W., Mambretti, A., Arshad, S., Weissbacher, M., Robertson, W., Kirda, E. and Egele, M.: HotFuzz: Discovering Algorithmic Denial-of-Service Vulnerabilities Through Guided Micro-Fuzzing, *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*, The Internet Society, (online), available from ⟨https://www.ndss-symposium.org/ndss-paper/hotfuzz-discovering-algorithmic-denial-of-service-vulnerabilities-through-guided-micro-fuzzing/⟩ (2020).

[15] Chen, H., Guo, S., Xue, Y., Sui, Y., Zhang, C., Li, Y., Wang, H. and Liu, Y.: MUZZ: Thread-aware Grey-box Fuzzing for Effective Bug Hunting in Multithreaded Programs, *29th USENIX Security Symposium (USENIX Security 20)*, USENIX Association, pp. 2325–2342 (online), available from ⟨`https://www.usenix.org/conference/usenixsecurity20/presentation/chen-hongxu`⟩ (2020).

[16] Xu, M., Kashyap, S., Zhao, H. and Kim, T.: Krace: Data Race Fuzzing for Kernel File Systems, *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 1643–1660 (online), DOI: 10.1109/SP40000.2020.00078 (2020).

[17] Österlund, S., Razavi, K., Bos, H. and Giuffrida, C.: ParmeSan: Sanitizer-guided Greybox Fuzzing, *29th USENIX Security Symposium (USENIX Security 20)*, USENIX Association, pp. 2289–2306 (online), available from ⟨`https://www.usenix.org/conference/usenixsecurity20/presentation/osterlund`⟩ (2020).

[18] Jeong, D. R., Kim, K., Shivakumar, B., Lee, B. and Shin, I.: Razzer: Finding Kernel Race Bugs through Fuzzing, *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 754–768 (online), DOI: 10.1109/SP.2019.00017 (2019).

[19] Kim, K., Jeong, D. R., Kim, C. H., Jang, Y., Shin, I. and Lee, B.: HFL: Hybrid Fuzzing on the Linux Kernel, *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*, The Internet Society, (online), available from ⟨https://www.ndss-symposium.org/ndss-paper/hfl-hybrid-fuzzing-on-the-linux-kernel/⟩ (2020).

[20] Pailoor, S., Aday, A. and Jana, S.: MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation, *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD, USENIX Association, pp. 729–743 (online), available from ⟨`https://www.usenix.org/conference/usenixsecurity18/presentation/pailoor`⟩ (2018).

[21] Dinh, S. T., Cho, H., Martin, K., Oest, A., Zeng, K., Kapravelos, A., Ahn, G., Bao, T., Wang, R., Doupé, A. and Shoshitaishvili, Y.: Favocado: Fuzzing the Binding Code of JavaScript Engines Using Semantically Correct Test Cases, *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*, The Internet Society, (online), available from ⟨https://www.ndss-symposium.org/ndss-paper/favocado-fuzzing-the-binding-code-of-javascript-engines-using-semantically-correct-test-cases/⟩ (2021).

[22] Agape, A.-A., Danceanu, M. C., Hansen, R. R. and Schmid, S.: P4Fuzz: Compiler Fuzzer ForDependable Programmable Dataplanes, *International Conference on Distributed Computing and Networking 2021*, ICDCN '21, New York, NY, USA, Association for Computing Machinery, p. 16–25 (online), DOI: 10.1145/3427796.3427798 (2021).