Vol.2019-OS-147 No.9
2019/7/25

ハイパーバイザを対象としたファジングへの効率的な初期シード生成手法の提案

石黒健太[1]　河野健二[1]

概要：ハイパーバイザの脆弱性はその上で動作するすべての仮想マシン (VM) のセキュリティを損なわせるため致命的なものである．ソフトウェアの脆弱性を自動で発見し，修正するためのテスト手法としてファジングが広く活用されているが，ハイパーバイザに適用することは簡単ではない．これは，ハイパーバイザとゲスト VM 間のインターフェースは VMEntry/Exit であることから入力がゲスト VM の状態となり，探索空間が膨大となるためである． ファジングの効率を向上させるための大きな要素として初期シードがある．しかし，ハイパーバイザ向けのファジングを行うために良い初期シードとして VM を構築することは，VM の状態が様々なレジスタおよびメモリの状態から構成されるため，難しい問題である．本論文では，ハイパーバイザを対象としたファジングの効率化のための初期シードの生成手法を提案する．現在，最先端の Linux カーネル向けのファジングツールである syzkaller では KVM をファジングするための初期シードの生成を手動で記述しているため，効率的なファジングにつながる VM の状態を作り出すことができていない． そこで，ゲスト VM 内で様々なワークロードを実行し，その際に得られるカバレッジをもとに効率的なファジングを可能とする初期シードとなる VM の状態を抽出する． 今回，KVM 向けのユニットテストをワークロードの例として活用し，初期シードの抽出方法によるファジングの効率への影響を確認する．

1. はじめに

ハイパーバイザはクラウドサービスを実現するための基盤ソフトウェアであり，Google App Engine や Amazon EC2 などの商用環境でも広く活用されている．バッファオーバフローや use-after-free に代表されるようなソフトウェアの脆弱性がハイパーバイザの中に存在している場合，ユーザ空間で動作するソフトウェアの脆弱性と比べ，致命的なものとなる．これは，ハイパーバイザの脆弱性がその上で動作するすべての仮想マシン (VM) のセキュリティを損なわせることが原因である．しかし，コード量の多さや考慮するアーキテクチャの複雑さ，さらにはテストすることの難しさからハイパーバイザの脆弱性は依然として数多く報告されている．例えば，幅広く使われているハイパーバイザの実装である KVM や Xen において，KVM では 2018 年 11 月までに 111 もの共通脆弱性識別子が登録されており，Xen では 275 もの脆弱性が Xen Security Advisories で報告されている．このように仮想化環境の重要性が高まっている一方で，その基盤ソフトウェアに対するテストは十分ではない．

ソフトウェアの脆弱性を自動で発見し，修正するためのテスト手法としてファジングが広く活用されている．現在ではユーザ空間のみならず，オペレーティング・システム (OS) に対してのファジングツールも活発に開発されている．しかし，これをハイパーバイザに対して直接適用することは簡単ではない．これは，ハイパーバイザとゲスト VM 間のインターフェースが VMEntry/Exit であることからファジングの際に与えるべき入力がゲスト VM の状態となり，探索空間が膨大となるためである．また，多くの OS 向けのファジングツールは OS とユーザ空間のアプリケーション間のインターフェースであるシステムコールをテストすることに注力している [4], [6]．このことから，ハイパーバイザのファジングに際して，最先端のファジングツールの提供している機能を最大限活用するためには VM を直接入力として用意するのではなく，システムコールの形式で入力を与えてゲスト VM を構築する必要がある．ハイパーバイザはホストのユーザ空間に対してシステムコールを提供することで自身やゲスト VM の状態・設定の変更を可能にしている．このシステムコールを用いることで OS 向けのファジングツールの機能を活用することが可能になっている一方で，ハイパーバイザの提供しているシステムコールは多岐に渡り，それらのシステムコール

[1] 慶應義塾大学
Keio Uniersity

の組み合わせでゲスト VM が構成されることから，提供されているシステムコールのシグネチャのみでは複雑なゲスト VM の状態を再現することは難しい．

OS 向けのファジングツールでは複数のシステムコールをある一つの順序にまとめたものを初期シードとし，その引数や順序を変異させていくことでコードカバレッジを向上させることを試みる．したがって，ハイパーバイザを対象としたファジングにおいては複数のシステムコールによって生成される VM とその VM に対する操作のシステムコールをあわせたものが初期シードとなる．初期シードの質および種類の豊富さはファジングツールのコードカバレッジを向上させる性能に大きく影響を与えることが知られている [11], [12], [15]．

本論文では，ハイパーバイザに対してのファジングの性能向上のための初期シードの生成手法を提案する．理想的な初期シードは，状態の変更が容易なように少ないシステムコールで構築されていながら，ハイパーバイザの多様な機能を実行するような VM の状態である．しかし，VM の状態が様々なレジスタおよびメモリの状態から構成されるため，システムコールをランダムに組み合わせるだけでは正しい VM の状態を作ることは難しい．正しくない状態の VM を用いたテストでは，ある特定のエラーハンドリングコードのみが実行され，ハイパーバイザのコア部分のコードカバレッジの向上は見込めない．最先端の OS 向けのファジングツールである syzkaller [4] ではこの問題を解決するために複数のシステムコールを組み合わせた擬似的なシステムコールの実装を手動で行っている．図 1 は疑似システムコール有効時および無効時におけるファジング実行時間に対するコードカバレッジを示している．疑似システムコール有効時は無効時と比べ，コードカバレッジが高くなっていることにくわえて，その飽和が遅くなっている．このことから，疑似システムコールは一定の効果をあげているが， VM の状態に対して網羅的なものではないため，依然としてハイパーバイザの多様な機能を実行するような初期シードの作成が必要である．

ハイパーバイザの多様な機能を実行できる VM の状態を用意するために，ゲスト VM 内で様々なワークロードを実行し，その際に得られるカバレッジを活用する．今回，KVM が用意するユニットテストをワークロードの例として使用し，ファジングの効率への影響を検討する．このユニットテストは KVM の開発者たちがハイパーバイザ内の特定の機能をテストするために用意したものであり，様々な特定な環境でのみ実行されるようなハイパーバイザの機能をテストできるような設計になっている．

実際，ユニットテストはファジングで到達することができなかったネストされた仮想化環境でのエミュレーション用のコードをカバーしており，これを初期シードとしてファジングツールに与えることによってカバレッジの向上につなげることが可能である．

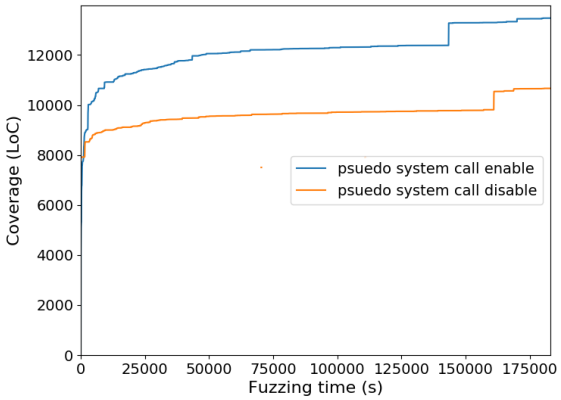

図 1 疑似システムコールの有効・無効時におけるファジング実行時間に対するコードカバレッジ

本論文の構成を以下に示す．第 2 章では，OS 向けファジングツールをハイパーバイザに適用する際の問題点を述べる．第 3 章では，本論文での提案であるユニットテストを用いた初期シード生成の方法を説明する．第 4 章では，提案手法の有用性を検証するために，実際に得られたコードカバレッジについて説明する．第 5 章では，本研究の関連研究について述べる．第 6 章では，まとめおよび今後の課題について述べる．

2. ハイパーバイザを対象としたファジングの問題点

現在，カバレッジを用いたファジングは American Fuzzy Lop (AFL) [1] や libFuzzer [3] に代表されるように様々なソフトウェアに対して効果を上げている．このようなファジングの戦略はユーザ空間のみならず OS を対象としたファジングでも効果を上げており，AFL を参考あるいはその実装に組み込んだ OS 向けファジングツールが多数公開されている [4], [5], [13] ハイパーバイザを対象としたファジングにおいても，これらの OS 向けファジングツールが用意しているカバレッジの収集やそれをもとにした新しい入力の生成などの様々な機能を活用することが望ましい．しかし，OS 向けのファジングではシステムコールをテストすることに注力していて，ゲスト VM が入力となるハイパーバイザには直接適用することができない．本章では，OS 向けのファジングツールをハイパーバイザの実装の一つである KVM に対して適用する方法とその際に生じる問題点について述べる．また，本論文の実験は，Intel Xeon Silver 4110 サーバ上で，ハイパーバイザとして Linux Kernel 4.18 の KVM および QEMU version 4.0.0，ファジングツールとして syzkaller を用いて行った．

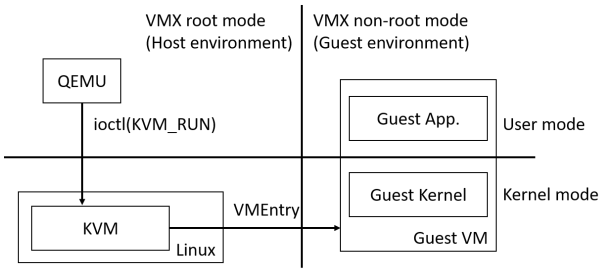

図 2　KVM/QEMU におけるゲスト VM を実行する流れ

2.1　KVM/QEMU のインターフェース

KVM は Linux に loadable kernel module (LKM) として組み込まれているハイパーバイザの一実装である．KVM は x86 環境において Intel VT-x や AMD-V と呼ばれるハードウェアによる仮想化支援機構を前提としている．Intel VT-x が有効な環境においては VMX root mode と VMX non-root mode という二つのモードが用意され，それぞれのモード内で ring protection の権限分離を行うことが可能となっている．

KVM では VM の作成やハイパーバイザの設定の変更のために QEMU などのユーザ空間のアプリケーションに対して API を用意している．この API は ioctl システムコールで実装されており，QEMU が ioctl システムコールを実行することで KVM が動作しゲスト VM を仮想環境上で動作させることを可能にしている．例えば VM を作成する場合には QEMU が `ioctl(KVM_CREATE_VM)` を実行することで KVM がオブジェクトを生成する．図 2 は QEMU からの要求を受けてゲスト VM が実行されるまでの処理の流れを示している．この図では QEMU があらかじめ作成した仮想 CPU を指定して `ioctl(KVM_RUN)` を実行することで，KVM が VMEntry し，仮想 CPU を VMX non-root mode 上で実行させている．このように VM の状態は API を通して管理することができるため，ユーザ空間のアプリケーションはシステムコールを通じて VM をセットアップすることができる．一方で KVM の API のドキュメントは 5263 行の自然言語で書かれている．そのため，手動でテストケースを作成していくことは一つ一つの API を理解している必要があり，熟練度の高くない開発者にとっては非常にコストが高いものとなる．

2.2　OS 向けファジングツールを用いる際の問題

ハイパーバイザを対象としたファジングを行う上で，カバレッジの効率的な収集やそれをもとにした新しい入力の作成といった機能を活用するために OS を対象としたファジングツールを用いることを考える．ハイパーバイザはゲスト VM が入力である一方，OS 向けファジングツールはシステムコールを対象とし，その引数や順番を変化させていくことでテストすることが通常であるため，一見，インターフェースが異なっている．しかし，第 2.1 節で述べたように，KVM はユーザ空間のアプリケーションに対して ioctl システムコールの形で API を提供しているため，ファジングツールもその API を用いてファジングを行うことが可能である．以降，この API を用いてファジングする際の問題点およびその問題点に対する現在既に実装されている解決手法とその問題点について説明する．

2.2.1　ゲスト VM の状態の豊富さによる問題

ファジングツールは KVM の API を入力として用いることができる一方で，KVM の API は多種多様に及ぶことにくわえて，VM のメモリやレジスタの種類は多岐にわたるため，正しい VM の状態をランダムなシステムコール列から生成することは難しい．正しい VM の状態を生成できない場合，ファジングを行った際の多くの時間がエラーハンドリングのロジックのテストに終始してしまい，ハイパーバイザのコア部分のコードをテストすることができなくなる．そこで，最先端の OS 向けファジングツールである syzkaller では，この問題を解決し，ハイパーバイザに対してのファジングの効率を高めるために，手動で仮想 CPU を設定する疑似システムコールを追加している．この疑似システムコールは複数のシステムコールをまとめたものとなっていて，これを実行することで仮想 CPU の状態を正しく設定する．したがって，疑似システムコールを初期シードに含めることによって，ハイパーバイザのコア部分のコードをテストすることが可能になる．

図 1 は疑似システムコールを初期シードに含めた場合と含めなかったの場合でのファジング時間に対するコードカバレッジを示している．同じファジング時間において疑似システムコールが有効の場合のほうが無効の場合と比較してコードカバレッジが高くなっていることにくわえて，コードカバレッジの上昇が飽和するまでの時間が遅くなっていることが確認できる．このように正しい VM の状態を初期シードとして含めることで，ハイパーバイザのコア部分をテストすることが可能になる一方で，手動によるシステムコールの追加では各種ハイパーバイザの機能を網羅的に設定することは難しい．これはゲスト VM のメモリやレジスタの状態は多岐にわたり，様々なモードを想定してハイパーバイザが実装されているため，特定の状況下において実行されるコードをすべて考慮することができないことが原因である．実際，ファジングによって得られたカバレッジを詳細に確認していくと，特定の状況において実行されるコードはカバーされておらず，2.2.2 項で述べるように，依然としてカバレッジが上がりにくい部分が残っていることが確認できる．このような状況を改善するために，ハイパーバイザの様々な機能を実行させることができる，特殊なケースを含む多様な初期シードを用意する必要がある．

2.2.2 ファジングでカバレッジを上げることができない状況の例

2.2.1 項で述べた通り，最先端の OS 向けファジングツールである syzkaller では，疑似システムコールを導入することで正しい VM の状態を作り出し，ハイパーバイザのコア部分のコードのテストを可能にした一方で，特定の状況において実行されるコード部のカバレッジは上がっていなかった．本項では，具体的にどのようなケースのコードがカバーされていないかを示し，またその原因を考察する．

まず，現状の疑似システムコールではネストされた仮想化環境向けのエミュレーション用のコードを実行することができていない．これは仮想 CPU をネストされた仮想化環境にて動かす場合には正常な VM の中にさらに正常な状態の VM を作成する必要があることが原因として考えられる．図 1 はゲスト VM が Control Register0 (cr0) 書き込むことで VMExit が起きた際のエミュレーションのコードを簡略化したものである．3 行目では VMExit した仮想 CPU がネストされた仮想化環境上で動作しているかどうかを判定している．ネストされた仮想化環境 (L2) で動作している場合，仮想環境上で動作しているハイパーバイザ (L1) のハイパーバイザを管理する構造体である Virtual Machine Control Structure (VMCS) を取得する (4 行目)．それと L2 が新たに設定しようと試みた cr0 の値から L1 に対して設定すべき cr0 の値を計算し，その値を L1 向けに設定する (5-6 行目)．最後にハイパーバイザが管理するシャドー cr0 レジスタに L2 が設定しようと試みた値を設定する (7 行目)．また，9 行目以降は仮想 CPU がネストされた仮想化環境以外の状況の場合に実行される．このようなコードの場合，9 行目以降はファジングによってテストすることができていたが，4-8 行目のネストされた仮想化環境用のエミュレーションのコードはテストすることができていなかった．

次に，ハードウェア設定を細かく決めることができておらず，カバレッジが向上していないケースも確認できる．図 2 は `syscall` 命令のエミュレーションを行うコードを簡略化したものである．3-5 行目および 7-8 行目では仮想 CPU の状態が `syscall` 命令を実行できる状態かどうかを確認し，不適切の場合，どちらも未定義命令による例外をゲスト VM 内で発生させる．仮想 CPU の状態が適切な場合，`syscall` 命令のエミュレーションを続行するが，今回の場合 7-8 行目の条件を満たすことができず，よりエミュレーションのコアな部分のテストをするには至っていなかった．

このように，手動で追加した疑似システムコールはコードカバレッジの向上に貢献しているものの，ゲスト VM の状態が様々なメモリやレジスタの状態から構成され，多岐に渡るため網羅的に作ることは難しく特定の状況をテストすることができていないということが確認できる．

```
1  static int handle_set_cr0(struct kvm_vcpu
       *vcpu, unsigned long val)
2  {
3    if (is_guest_mode(vcpu)) {
4      struct vmcs12 *vmcs12 = get_vmcs12(
           vcpu);
5      unsigned long cr0_l1 = value_for_l1(
           vmcs, val);
6      kvm_set_cr0(vcpu, cr0_l1);
7      vmcs_write(CR0_READ_SHADOW, val);
8      return 0;
9    } else {
10     return kvm_set_cr0(vcpu, val);
11   }
12 }
```

コード 1 cr0 レジスタへの書き込みをエミュレーションするためのハンドラ

```
1  static int em_syscall(struct
       x86_emulate_ctxt *ctxt)
2  {
3    if (ctxt->mode == X86EMUL_MODE_REAL ||
4        ctxt->mode == X86EMUL_MODE_VM86)
5      return emulate_ud(ctxt);
6
7    if (!(em_syscall_is_enabled(ctxt)))
8      return emulate_ud(ctxt);
9
10   /* emulation for syscall instruction */
11 }
```

コード 2 syscall 命令のエミュレーション

3. 提案及び実装

3.1 概要

ハイパーバイザの多種多様な機能を実行するように様々な種類の初期シードを生成することは，ハイパーバイザを対象としたファジングの効率を向上させる．このような多様な初期シードを生成するために，様々なワークロードを VM 上に用意し，実行時のカバレッジ調査することでファジングの効率を向上させる VM の状態を抽出する．現在のファジング手法では，2.2.2 項で述べたように，ハイパーバイザの特定の状況でのみ実行されるような機能をテストすることに適していない．そのため，ファジングの効率を向上させるような VM の状態として，その特定の状況でのみ実行されるコードを実行する VM を抽出し，それを初期シードとして活用できるようにすることで，特定の環境下で実行されるコードのテストを可能にする．

初期シードとなる VM の状態を抽出するにあたって，次の二点を考慮する必要がある．

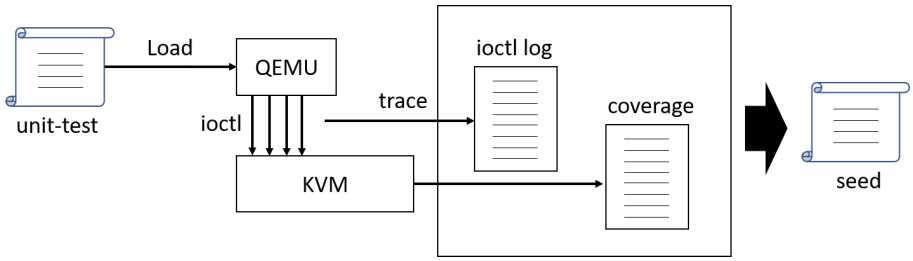

図 3 ユニットテストを用いた初期シード生成方法

- どのワークロードがハイパーバイザの多種多様な機能を実行しているかということの判定
- ワークロードが作り出す VM の状態の再現方法

本提案手法では，ハイパーバイザの多種多様な機能を実行しているかどうかを判定するために，ワークロード実行時のカバレッジを用いる．ワークロード実行時のカバレッジとファジングで得られたカバレッジを比較することで，新たなコードカバレッジを生み出しているワークロードを見つけだす．このとき，そのワークロードが特定状況でのみ実行されるコードをカバーできるような VM の状態を作り出していると判定する．また，ワークロードが作り出す VM の状態を再現するためにワークロードが呼び出す KVM の API の順序とそれぞれの引数を記録する．

3.2 実装

今回，VM 上のワークロードのとして KVM 向けのユニットテスト [2] を活用した．このユニットテストは C 言語とアセンブリ言語で書かれたコードから小さなゲストカーネルを生成し，KVM 上で実行することで KVM や仮想ハードウェアのテストを行う．また，ワークロード実行時のカバレッジの情報は kcov を用いて取得し，QEMU の ioctl システムコールの実行をトレースすることで，ワークロードが呼び出す KVM の API の順序とそれぞれの引数の情報を取得した．図 3 は初期シード生成の流れを示している．ユニットテストは小さなゲストカーネルとして QEMU にロードされる．このゲストカーネルを QEMU は仮想化環境で動作させるために ioctl を呼び出し KVM とインタラクトを行う．そして，このユニットテスト実行時に取得できたカバレッジの情報から，通常のファジングではテストすることのできないコードを実行しているテストケースを抽出し，そのテストケースの ioctl の実行ログを初期シードとして活用する．

4. 実験

今回は予備実験として，x86 用に用意されているユニットテストを実行し，そのカバレッジを取得した．その後，2.2.2 項で述べたファジングでテストできていなかったケースと比較し，ユニットテストを初期シードとしたファジングが効果的になりうるかどうかを調査した．

図 1 はネストされた仮想化環境でのエミュレーションのコードであるが，ユニットテストのうち，ネストされた仮想化のためのユニットテストではこのエミュレーションをカバーしていた．そのため，この部分の KVM の API の実行ログを初期シードとして用いることで，通常のファジングのみではテストできなかった部分をカバーすることが可能となる．

一方で図 2 は `syscall` 命令のエミュレーションのためのコードであるが，ユニットテストではこの部分をカバーするテストは存在しなかった．

このように，ユニットテストをワークロードとした初期シードの生成は一定の効果を上げる一方で，さらなるファジング効率の向上にはユニットテスト以外のワークロードの使用を試みる必要がある．

5. 関連研究

ハイパーバイザを対象としたテスト

ハイパーバイザの信頼性向上のためにテストケースを生成することは様々な研究で行われている．Virtual CPU Validation [7] ではいくつかのハイパーバイザの脆弱性を Intel の物理 CPU 向けのテストケースを KVM 対して適用することで発見している．この手法は，実際のハイパーバイザのコードを分析せずにテストケースを生成している．本論文では既存の OS 向けファジングツールを用いることでハイパーバイザのコードカバレッジを効果的に活用することを可能としている点で異なる．MultiNyx [9] は命令エミュレータである Bochs を活用し，仮想化に用いられる複雑な命令を解析することで KVM 向けのシンボリック実行を用いたテストケースの自動生成を行っている．さらに，テストケースの実行結果を異なる複数の環境で実行することで比較し，不一致となる例を発見している．

実際のアプリケーションの実行結果を用いた OS を対象としたファジング手法

OS を対象としたファジングにおいて，良い初期シードを生成することは簡単ではない．それはシステムコール実行時の挙動はそれ以前に実行されたシステムコールによって変更された OS の状態に強く依存するためである．IMF [10] はモデルベースのカーネルファジングツールである．様々なアプリケーションを macOS 上で動かし，カーネル API

呼び出しのトレースからファジングの入力となるモデルを生成することでシステムコールの依存関係の問題を解決している．MoonShine [11] では初期シードのサイズを小さくすることで初期シードの質を高め，ファジングの効率を上げている．実際のアプリケーションを実行した際に得られるトレースはサイズが大きくなり，そのまま用いるのでは新たな入力を作る際の必要な時間が大きくなりファジングの性能に悪影響を及ぼす．そこで，アプリケーション実行時に得られたトレースから暗黙および明示的な依存関係を明らかにすることで，初期シードを実行した際に得られるカバレッジを損なわずに余計なシステムコールを取り除いている．

カーネルの提供しないインターフェースを用いたファジング手法

OS を対象としたファジングツールはシステムコールの引数や順番を入れ替えることでコードカバレッジを向上させている．しかし，ハイパーバイザやデバイスドライバのように独自でシステムコールとして API を提供している場合にはその入力の複雑さから正しい内部状態を作り出すことが難しくなっている．DIFUZE [8] では，デバイスドライバを効果的にファジングするために，デバイスドライバのコードに対して静的解析を行い，デバイスドライバのインターフェースを解析することで，ユーザ空間からデバイスドライバを呼び出す際の正しい入力を生成している．PeriScope [14] はデバイスドライバとデバイス間のインターフェースに注目し，システムコールを用いない周辺機器からの入力に対してのデバイスドライバの信頼性を向上させることを目的としている．そのため，デバイスドライバとデバイスのインタラクトを監視し，デバイスからのデータストリームをファジングツールを用いて変化させていくことでデバイスドライバに対してファジングを行っている．

6. まとめ

ハイパーバイザを対象としたファジングでは，最先端の OS 向けファジングツールを適用した際に，入力となるゲスト VM の取りうる状態の多さから正しい入力を作ることができず，ファジングの効率が上がらないという問題点がある．また，それを解決するための正しい初期状態を生成する入力は効果的である一方で，手動で実装されているために網羅的に実装するのは難しく，特定の状況下で実行されるようなコードをテストすることができない．そこで本論文では，ハイパーバイザを対象としたファジングの効率を向上させるために，ゲスト VM 上で様々なワークロードを実行し，その際に得られるカバレッジをもととした初期シードの生成手法を提案した．これにより，手動の疑似システムコールでは網羅できていないネストされた仮想化のためのエミュレーションコードに対してコードカバレッジを上げるような入力を生成することができた．

謝辞 本研究は，JST，CREST，JPMJCR19F3 の支援，JSPS 科研費 JP19K11906 の助成を受けたものである．

参考文献

[1] : afl, `http://lcamtuf.coredump.cx/afl/`.
[2] : KVM-unit-tests, `https://www.linux-kvm.org/page/KVM-unit-tests`.
[3] : libFuzzer, `http://llvm.org/docs/LibFuzzer.html`.
[4] : syzkaller, `https://github.com/google/syzkaller`.
[5] : TriforceLinuxSyscallFuzzer, `https://github.com/nccgroup/TriforceLinuxSyscallFuzzer`.
[6] : trinity, `https://github.com/kernelslacker/trinity`.
[7] 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, ACM, pp. 311–327 (online), DOI: 10.1145/2815400.2815420 (2015).
[8] Corina, J., Machiry, A., Salls, C., Shoshitaishvili, Y., Hao, S., Kruegel, C. and Vigna, G.: DIFUZE: Interface Aware Fuzzing for Kernel Drivers, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, New York, NY, USA, ACM, pp. 2123–2138 (2017).
[9] Fonseca, P., Wang, X. and Krishnamurthy, A.: MultiNyx: A Multi-level Abstraction Framework for Systematic Analysis of Hypervisors, *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, ACM, pp. 23:1–23:12 (online), DOI: 10.1145/3190508.3190529 (2018).
[10] Han, H. and Cha, S. K.: IMF: Inferred Model-based Fuzzer, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ACM, pp. 2345–2358 (2017).
[11] Pailoor, S., Aday, A. and Jana, S.: MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation, *27th USENIX Security Symposium (USENIX Security 18)*, pp. 729–743 (2018).
[12] Rebert, A., Cha, S. K., Avgerinos, T., Foote, J., Warren, D., Grieco, G. and Brumley, D.: Optimizing Seed Selection for Fuzzing, *23rd USENIX Security Symposium (USENIX Security 14)*, pp. 861–875 (2014).
[13] 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)*, pp. 167–182 (2017).
[14] Song, D., Hetzelt, F., Das, D., Spensky, C., Na, Y., Volckaert, S., Vigna, G., Kruegel, C., Seifert, J.-P. and Franz, M.: PeriScope: An Effective Probing and Fuzzing Framework for the Hardware-OS Boundary, *Proceedings 2019 Network and Distributed System Security Symposium*, NDSS '19 (2019).
[15] Wang, J., Chen, B., Wei, L. and Liu, Y.: Skyfire: Data-Driven Seed Generation for Fuzzing, *2017 IEEE Symposium on Security and Privacy (SP)*, pp. 579–594 (2017).