

ユーザの支援を用いたファジングによる ハードウェアテスト

杉山 優一^{1,a)} 塩谷 亮太¹

概要：ハードウェアのバグや脆弱性は発見が難しい一方で、製造後の修正も極めて困難である。このため、そのようなバグや脆弱性を事前に見つけるためのテスト手法が多く提案されてきた。それらの手法は大きく分けて、手動または自動で生成した入力を用いる動的な手法と、形式的手法を用いた静的な手法に分けられる。これに対し、大規模なハードウェアでは動的な手法は高いカバレッジを実現するために極めて長い時間がかかることが多い。その一方で、静的な手法は組み合わせ爆発によりそもそも有効な検証が行えないことが多い。これらの問題に対し、ソフトウェア分野におけるファジングと呼ばれるテスト技術をハードウェアのテストに応用した手法が研究されている。ファジングは基本的には動的な手法であり、一般に、実行時カバレッジなどの情報をフィードバックしながらテスト対象の状態空間を探索する。これにより、小さなユーザの労力で効果的なテスト入力を生成することができる。しかし、既存のハードウェアにおけるファジングには、ハードウェアの状態空間の探索方法が良くないという問題がある。これに対し、本論文では (1) カバレッジ指標と (2) 探索方向のそれぞれについて、ユーザの支援を用いることで状態空間を効率的に探索できる手法を提案する。提案手法では、既存手法のようにハードウェア設計の構造から自動で解析できるカバレッジ指標ではなく、ユーザが選択した重要なレジスタをカバレッジ指標として使う。また提案手法では、既存手法のようにハードウェア全体を均一に探索するのではなく、ユーザの作成した探索方向の優先度を示す指標に基づき指向性のある探索をする。比較的規模が大きく複雑なプロセッサ設計を対象に提案手法を適用した結果、既存手法では発見が難しい未知のバグを発見することができた。

1. はじめに

ハードウェアのバグや脆弱性は発見が難しい一方で製造後の修正も極めて困難であり、多くのバグや脆弱性が深刻な問題を起こしてきた。ハードウェアの中でも特にプロセッサでは後からバグを修正することが難しく、かつ極めて広範囲のアプリケーションに問題を与えるため、バグや脆弱性による影響が一層深刻である [1]。たとえば、商用マイクロ・プロセッサである Pentium では、特定の値の除算の結果が誤ったものになるという浮動小数点演算器のバグを持っていた。このバグは修正が困難であり、最終的にはプロセッサの無償交換が行われるなど大きな損害を発生させた。[2]。さらに最近では、プロセッサの投機的実行に関連する脆弱性である Spectre [3] や Meltdown [4] が発見され、それらがソフトウェアのセキュリティを保つ上で重大な課題となっている。このようにハードウェアのバグや脆弱性の影響は深刻であり、それを解決する有力な手段である様々なハードウェアのテスト技術が注目されている。

多くのハードウェアのテスト技術では、なるべく広い範囲のハードウェアの状態をテストし、バグや脆弱性ないことを確認することを目指す [5], [6], [7], [8], [9], [10], [11], [12]。それらは大きく分けて、形式的手法を用いた静的な手法 [5], [6] と、手動またはランダムに作成したテスト入力をを用いた動的な手法 [7], [8], [9], [10], [11], [12] に分けられる。静的な手法はモデル検査などを用いて、テスト対象が何らかの特性を満たしているかどうか証明することができる。たとえば、検証可能な特性には、デッドロック起きないことなどが挙げられる。しかし、静的な手法は組み合わせ爆発により大規模なハードウェアに対しては現実的な計算時間でそもそも有効な検証が行えないことが多い。一方で、動的な手法は大規模なハードウェアに対しても現実的な計算時間で適用可能な場合が多い。しかし、高いカバレッジを実現するためには多くの有効なテストを作る必要があり、多くの手動による作業や極めて長い時間を要してしまう。

このような問題に対し、**ファジング**と呼ばれるソフトウェア分野におけるテスト技術を、ハードウェアのテストに応用した手法が研究されている [13], [14], [15], [16], [17]。

¹ 東京大学 大学院情報理工学系研究科

^{a)} sugiyama@rsg.ci.i.u-tokyo.ac.jp

ファジングは基本的には動的な手法であり、一般に、実行時のカバレッジなどの情報をフィードバックしながらテスト対象の状態空間を探索する。これにより、動的な手法の問題を解決し、小さなユーザの労力で効果的なテスト入力を生成することができる。代表的なハードウェアにおけるファジングの既存研究である RFUZZ [13] では、抽象化された回路構造を表すモデル上のマルチプレクサの選択信号をカバレッジ指標として用いている。また DIFUZZRTL [14] では、マルチプレクサの選択信号に接続されている制御レジスタをカバレッジ指標として用いることで、より効率的な状態空間の探索を実現している。

しかし、これらの既存手法が用いているカバレッジ指標では、ハードウェアの状態空間を適切に扱うことができないため、効率的な探索を実現できていない。多くの既存研究では、抽象化された回路構造を表すモデルから機械的に決定できるカバレッジ指標を用いている [13], [14]。このようなカバレッジ指標は自動で定義できるが、カバレッジ指標として扱う状態数の減らし方に問題がある。そのため、既存手法では状態空間内のバグの近傍を十分に探索できない場合や、バグの近傍から外れたところを探索してしまう場合がある。

そこで、本論文では (1) カバレッジ指標と (2) 探索方向のそれぞれについて、ユーザの支援を用いることで状態空間を効率的に探索する手法を提案する。提案手法では、テストを行うユーザやハードウェア設計者が経験的にバグが発生しやすい箇所や状況を知っていることが多いことを利用する。たとえば、バッファが一杯の場合の例外処理や、テーブルにある同一エントリへの並列アクセスなどは、処理が複雑でバグが起きやすい。また、ソースコード変更時のバグ混入は一般に変更部分の近辺で発生する事が多い [18]。そこで、そのような知識に基づくユーザの支援を用いて、より適切な探索空間と探索方向を設定し、ユーザが指定した部分を集中的に探索する。

提案手法を実際の回路を模した様々なハードウェア設計を用いて評価したところ、既存手法と比較して大幅に短い時間で意図的に挿入したバグを発見でき、状態空間を効率的に探索できることを示した。また、オープンソースの比較的大きくで複雑なプロセッサ設計である RSD [19] に対して評価を行った。その結果、提案手法は既存手法では発見が難しい未知のバグを発見することができた。

本論文の構成は以下の通りである。まず 2 節で背景として、ソフトウェアにおけるファジングについて述べる。3 節でハードウェアにおけるファジングの既存研究とその問題について述べる。4 節では本論文が提案するユーザの支援を用いたハードウェアにおけるファジングについて述べ、5 節では提案手法の評価を行う。6 節で関連研究を示し、最後に 7 節でまとめると今後の課題について述べる。

2. ソフトウェアにおけるファジング

ファジング (ファズテスト) とは、主にソフトウェアのテストを対象としたテスト技術の一つである。ファジングでは対象となるプログラムに様々な入力を与えて、バグや脆弱性を発見することを目的とする [20], [21]。これは基本的には動的な手法であり、実行時のテスト対象の情報をフィードバックしながら、入力の生成とテスト対象の実行を繰り返す。これにより、テストを行うユーザが手動で入力を生成することなく、テスト対象のプログラムの状態を効率的に幅広くテストするための入力を自動で生成できる。実際に、多くのバグや脆弱性がファジングを用いて発見されている [22], [23]。

ここでテスト対象のプログラムの状態とはレジスタやメモリなどの状態のことであり、状態空間とはプログラムが取り得るすべての状態の集合である。一般にプログラムの状態空間は極めて大きく、状態空間を全て探索することは現実的ではない。そのため、一般にファジングでは状態空間そのものを直接探索するのではなく、カバレッジと呼ばれる指標に基づいて探索を行う。カバレッジとは、何らかの網羅条件がテストによってどれだけ実行されたかを表すものである。たとえば、ソフトウェアにおけるファジングのカバレッジでは、コード中の実行済みの行や分岐方向の網羅率が用いられる。ファジングでは、このような何らかのカバレッジを高めるように探索することで、プログラムの状態空間を効率的に探索する。このように、ファジングはカバレッジを用いてプログラムの状態空間を探索するため、カバレッジ指標はファジングの性能を決める重要な要因の一つとなっている [24]。

カバレッジに基づくファジングは一般に、アルゴリズム 1 に示す 6 つの段階で構成される。テストを行うユーザはファザーに事前に用意された入力集合である S と、ファジングの総時間 t_{limit} を与える (S1)。次に、ファジングを行うプログラムは、シードの集合 S から入力生成のシードとなる入力 s を一つ選び (S2)、入力 s の一部を変異させたテスト入力 s' を作成する (S3)。最後に、ファジングを行うプログラムはテスト入力 s' を用いて、テスト対象のハードウェアを実行する (S4)。それと同時に、ファジングを行うプログラムは実行時のクラッシュなどを確認し、クラッシュした入力 s' を C に保存する (S5)。また、カバレッジが高くなるテスト入力 s' は重要であると認識され、 S に追加される (S6)。これにより、次回以降その重要な入力を変異させて、新たな入力を生成できる。ファジングでは、フィードバック・ループを用いて、(S2) から (S6) をファジングの総時間 t_{limit} 繰り返し行い、カバレッジが高くなるように状態空間を探索する。

アルゴリズム 1 カバレッジに基づくファジング

Input: (S1) Seed Inputs S , Total Fuzzing Duration t_{limit}

Output: Crashing Inputs C

$C = \emptyset$

while $t_{elapsed} < t_{limit}$ **do**

(S2) $s = ChooseNext(S)$

(S3) $s' = Mutate(s)$

(S4) $o = Execute(s')$

if $IsCrashing(o)$ **then**

(S5) $C.add(s')$

else if $IsInteresting(o)$ **then**

(S6) $S.add(s')$

end if

end while

3. ハードウェアにおけるファジング

ハードウェアにおけるファジングとは、ソフトウェアにおけるファジングをハードウェアのテストに応用した技術である。ハードウェアにおけるファジングでは、従来の動的なハードウェアのテスト技術の問題を解決し、小さな労力で効果的なテストを行うことを目指している。ランダムに生成したテスト入力や手作業で作成したテスト入力を用いて、高いテストカバレッジを達成するには多くの時間やユーザの労力が必要である [7], [8]。これに対し、2 節で説明したように、ファジングでは実行時のカバレッジなどのフィードバックを用いることで、有効なテスト入力を効率的に自動で生成できる。ハードウェアにおけるファジングは、このようなファジングの利点を用いて、従来のハードウェアのテスト技術の問題解決を目指す。

ハードウェアにおけるファジングのアルゴリズムとソフトウェアにおけるファジングのアルゴリズムの大きな違いの一つは、アルゴリズム 1 の (S6) のテスト入力の評価で用いられるカバレッジ指標である。ファジングでは、カバレッジのフィードバックを用いて、各テスト入力が到達した探索空間内の状態を認識する。そのため、カバレッジ指標が適切にテスト入力が到達した状態を認識し、効果的なフィードバックを返すことがとても重要である。本節では、ナイーブなハードウェアにおけるカバレッジ指標と、ファジングの代表的な 2 つの既存研究が提案しているカバレッジ指標とその問題点について説明する。

3.1 ナイーブなカバレッジ指標

本研究ではハードウェアの状態を、それに含まれる全てのレジスタの状態であると考え、この場合、ナイーブなハードウェアのカバレッジは、全レジスタの状態の網羅率となる。そのため、全レジスタのビット数の合計が N であるとき、カバレッジ指標が扱う状態数は $O(2^N)$ になる。

一般にハードウェア設計に含まれるレジスタのビット数は莫大であるため、ナイーブなカバレッジ指標が扱う状態

表 1: RISC-V CPU に内蔵されるレジスタとマルチプレクサの数

	レジスタ のビット数	マルチプレクサ の数	制御レジスタ のビット数
Rocket [25]	15300	5300	661
Boom [26]	36600	21000	990

数は爆発してしまう。表 1 は、2 つの RISC-V CPU に含まれているレジスタとマルチプレクサの数を示している。たとえば、RISC-V CPU の Boom のレジスタのビット数は、36600 ビットあるため、状態数は 2^{36600} になる。このようにナイーブな方法では探索すべき状態数が爆発し、現実的に扱うことができない。

このため、ハードウェアにおけるファジングでは一般に、状態爆発を避けるような別の方法に基づいて探索を行う [13], [14]。以下ではそのような探索を行う 2 つの手法について、順に説明する。

3.2 マルチプレクサの選択信号に基づくカバレッジ指標

RFUZZ [13] は、ハードウェアの状態を記憶するためのレジスタではなく、マルチプレクサの選択信号に基づくカバレッジ指標を提案している。マルチプレクサの選択信号は、マルチプレクサの入力のどちらを出力するかを決めるための信号であり、その網羅率を観測する。マルチプレクサは HDL (Hardware Description Language) で記述された分岐として合成されることが多い。マルチプレクサの選択信号を観測することは、HDL 上の各行が活性化された部分を間接的に観測していることに近い意味を持つ。また、マルチプレクサの選択信号の数は一般にレジスタより少ないため、ナイーブなカバレッジ指標より見るべき対象を減らすことができる。たとえば、表 1 に示すように、RISC-V CPU の Boom [26] ではレジスタのビット数は 36600 であるのに対し、マルチプレクサの数は 21000 と大幅に少ない。

マルチプレクサの数はレジスタのビット数より少ないが、しかしそれでもなお膨大である。そのため、RFUZZ が提案するカバレッジ指標では、マルチプレクサの選択信号の組み合わせを考慮しない。つまり、マルチプレクサの数が N であるとき、各マルチプレクサの選択信号が独立に活性化されたかを確認することになる。そのため、扱う状態数を $O(2^N)$ から $O(N)$ に削減できる。しかし、このように組み合わせを考えないと、選択信号の特定の組み合わせで起きるバグをテストできないことが知られている [14]。

3.3 制御レジスタに基づくカバレッジ指標

DIFUZZRTL [14] では RFUZZ [13] より見るべき対象を減らした上で、組み合わせを考慮できるカバレッジ指標を提案している。DIFUZZRTL はカバレッジ指標としてマルチプレクサの選択信号の代わりに、制御レジスタを用いることを提案している。制御レジスタとは、マルチプレク

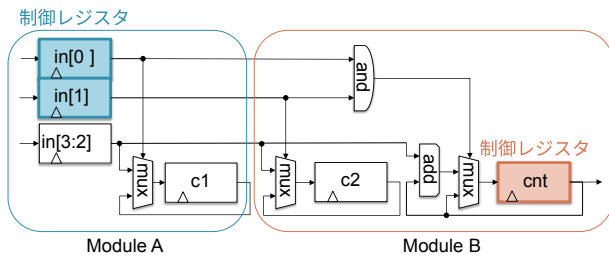


図 1: 問題 1: 複数のモジュールの状態の組み合わせを認識できない

サの制御信号に接続されるレジスタのことである。一般に一つの制御レジスタは複数のマルチプレクサの選択信号に接続されるため、制御レジスタのビット数はマルチプレクサの数より少なくなる。たとえば、表 1 に示すように、RISC-V CPU の Boom [26] ではマルチプレクサの数は 21000 であるのに対し、制御レジスタのビット数は 990 と大幅に少ない。

しかし、一般にハードウェア設計内には多くの制御レジスタが内蔵されるため、全ての制御レジスタの組み合わせをカバレッジとして定義することは難しい。これは単純に全ての制御レジスタの組み合わせをカバレッジとして考えると、状態数が爆発してしまうことが多いためである。そこで、DIFUZZRTL はモジュール境界を利用して、制御レジスタの組み合わせを分割している。たとえば、Boom に内蔵される 990 個の制御レジスタが、100 個の各モジュールに均等に配置されている場合を考える。全体の組み合わせを考えると、カバレッジとして定義される状態数は 2^{990} になる。一方で各モジュール単位で組み合わせを考えた場合、状態数は $100 \times 2^{9.9}$ になる。このように、モジュール単位の組み合わせを用いることで状態数を減らし、現実的に計算可能なカバレッジ指標を定義している。

DIFUZZRTL は RFUZZ に比べて高い性能を示しているが [14]、制御レジスタに基づくカバレッジ指標には 2 つの問題がある。1 つ目の問題は、複数のモジュールの状態の組み合わせを正確に認識することができず、複数のモジュールに含まれるレジスタの状態の組み合わせを効率的にテストできないことである。モジュールはハードウェア設計者が記述しやすいように作成するものであり、必ずしも意味的に関連するもの同士がまとめられているわけではない。たとえば、1 つの機能が分散されて実装されることもある。そのため、各モジュールごとの組み合わせのみを考えると、関連するモジュールの状態の組み合わせを効率的にテストできない。図 1 に示す回路では、cnt レジスタと、それにデータを書き込むかどうかを決める 2 つレジスタ in0 と in1 が別々のモジュールに含まれている。そのため、DIFUZZRTL では、これら 3 つの状態の組み合わせを考慮することができない。しかし、in0 と in1 の状態は cnt の状態を決める要因となっているため、これらはグループ化して、組み合わせを考慮すべきである。

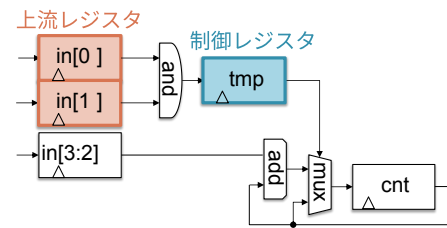


図 2: 問題 2: 上流のレジスタの状態を認識できない

2 つ目の問題は、間接的に動作に影響を与えている上流のレジスタの状態を認識できないことである。DIFUZZRTL はマルチプレクサの選択信号に直接接続されているレジスタのみをカバレッジ指標として用いる。これは考慮するレジスタの数を減らすためである。しかし、これにより、間接的に接続された上流のレジスタの状態を探索できない。これは特に、単に信号を中継する目的で挿入したパイプライン・レジスタなどで問題となる。たとえば、図 2 に示す回路は、in0 と in1 の論理演算の結果を一度 tmp レジスタに格納し、tmp レジスタの値がマルチプレクサの選択信号に使われていることを示している。このとき、tmp レジスタは制御レジスタになるが、上流にある 2 つのレジスタは制御レジスタとして認識されない。一般に上流にあるレジスタのほうがより多くの情報を持つことが多いため、それらをカバレッジとして見ないのは問題である。

4. ユーザの支援を用いたハードウェアにおけるファジング

既存手法の問題を解決するために、我々はユーザの支援を用いたハードウェアにおけるファジングを提案する。状態空間の中から、ユーザが指定した部分を集中的に探索することで、バグを効率的に発見する。設計者は経験的にバグが発生しやすい箇所や状況を知っていることが多い。たとえば、バッファが一杯の場合の例外処理や、テーブルにある同一エントリへの並列アクセスなどは、処理が複雑でバグが起きやすい。また、バグはソースコードの変更部分で新たに発生することもある [18]。たとえば、CVE-2016-5728 は不注意なコードの変更によって発生した。したがって、バグの発生しやすい箇所や状況、または変更部分を集中的にテストすることで、バグを検出する確率が高くなりうる。本節ではこの考え方にに基づき、2 つの提案手法について説明する。

4.1 ユーザの支援を用いたカバレッジ指標の設定

1 つ目の提案は、ユーザの支援を用いたカバレッジ指標の設定である。提案手法では、カバレッジとして、ユーザの選んだレジスタと、それに関連するレジスタの組み合わせを用いる。まず、ユーザは重要なレジスタを選択する。たとえば、キューの使用量を格納するレジスタや、コミッ

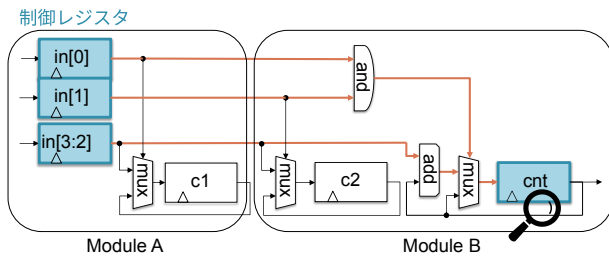


図 3: ユーザによるレジスタの選択とデータフロー解析

ト・ログなどから分かる新しく追加されたレジスタ*1などを選択する。これらのレジスタでは他の部分と比べて高い確率でのバグの発生が予想される。このため、これらのレジスタの状態を集中的に探索することで高い確率でバグを発見できることが期待できる。次に、データフロー解析を用いて、そのレジスタと関連のあるレジスタを検出する。この2つのステップにより、ユーザの選択したレジスタと、その状態を決める要因となるレジスタの組み合わせからなるカバレッジ指標を作成できる。たとえば、図3に示す回路は図1と同じ回路に対して、提案手法を適用した場合を示している。提案手法では cnt レジスタをユーザが選択し、モジュール境界を越えたデータフロー解析により、cnt の状態を決める in レジスタを検出している。この2つのステップにより、ユーザの選択したレジスタと、その状態を決める要因となるレジスタの組み合わせからなるカバレッジ指標を作成できる。

このようなカバレッジ指標を用いることで、少ない状態数のまま、モジュール境界に縛られず、真に重要な部分のレジスタの組み合わせを探索できる。ユーザの指定したレジスタの状態は上流のレジスタの状態によって決まる。このため、回路構造に基づき、ユーザの選択したレジスタと上流レジスタをグループ化して、それらの組み合わせをみるのが重要である。このようにすることで、真に重要な部分のレジスタの組み合わせに関する情報をカバレッジのフィードバックとして得られるようになる。

4.2 ユーザの支援を用いた探索方向の設定

2つ目の提案手法はユーザの支援を用いた探索方向の設定である。ユーザは選択したレジスタに対して、探索方向の優先度を設定する。提案手法では、この優先度に基づき、よりバグの起きやすい方向に向かって探索を進めることで、バグを効率的に発見する。一般に、キューが埋まっている場合や調停における競合発生の場合など、例外的な処理を行う場合には論理が複雑となりがちである。また、これらの例外的な状況は通常の処理ではなかなか発生しないため、その検証も困難で有ることが多い。そのため、このような状況ではバグが発生しやすく、そのような状態に対

して高い優先度を割り当てることができる。そして、その優先度に基づき、高い優先度の入力をより多く変異させることで、ユーザが指定した状態に向かって指向的に探索を進めることができる。

5. 評価

本論文では様々な設計を用いて、提案手法の性能を評価した。5.1 節では評価環境について述べる。5.2 節では人工的に作成した2つのハードウェア設計による評価結果について述べ、5.3 節では実際のハードウェア設計による評価結果について述べる。

5.1 評価環境

提案手法と既存手法の評価を行った。評価モデルは以下の3つである。

- (1) random: フィードバックを用いない完全にランダムな入力生成
- (2) difuzzrtl: 制御レジスタに基づくカバレッジを用いたファジング [14]
- (3) proposal: 各ハードウェア設計向けのユーザの支援を用いたファジング

評価では以下で述べるように、人工的に作成したハードウェア設計と実際のハードウェア設計を用いた。

まず、人工的に作成したハードウェア設計による評価では、以下で述べる小規模なハードウェアにより、既存手法や提案手法の詳細な振る舞いを比較し解析した。

- (1) 人工的に作成したハードウェア設計 1: テスト対象として DIFUZZRTL [14] の評価で用いられている評価用のハードウェア設計を一部改変したものをを用いた。3つの各モジュールに状態機械を内蔵するハードウェアであり、メモリコントローラを模している。また意図的にバグが挿入されており、状態機械が特定の状態の組み合わせになった場合にバグが発生する。提案手法では、ユーザの支援として、状態機械の状態を格納するレジスタを指定する。
- (2) 人工的に作成したハードウェア設計 2: テスト対象としてオープンソースの CPU の中で使われているキューを用いた。提案手法に対するユーザの支援として、キューの使用量を格納するレジスタの選択と、キューの使用量が大きくなる方向に優先度を設定した。

次に、実際のハードウェア設計による評価では、大規模な設計において提案手法が有効に働くかどうかを評価した。ここでは、比較的規模が大きく複雑なプロセッサ設計である RSD [19] を用いた。RSD は 32 ビットの RISC-V アウ

*1 この変更があったレジスタの抽出は自動化できるため、ユーザによる直接的な支援は必要ない。

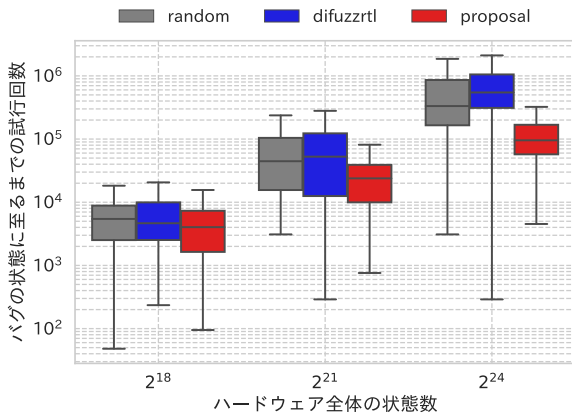


図 4: 3つの状態機械を内蔵するハードウェア設計に対する結果の分布

ト・オブ・オーダー実行のスーパー・スカラ・プロセッサであり、オープンソースであり様々な研究において試作や評価に使われている [27], [28], [29]. また RSD は、既に手動で作成した各種の入力を用いて検証されている. この評価ではテスト対象がプロセッサであるため、入力の変異ではアセンブリ命令の変異を実装している. 具体的には、オペコードとオペランドの変異、命令の入れ替えなどを行う. また、ユーザの支援は HDL コードへの注釈によって行われる. 評価の結果、提案手法では未知のバグを発見できたため、それに必要な時間も評価を行った.

5.2 人工的に作成したハードウェア設計による評価の結果

5.2.1 3つの状態機械を内蔵するハードウェア設計

3つの状態機械を内蔵するハードウェア設計を用いた実験では、意図的に挿入したバグを発生させる入力を作成できるまでの試行回数を評価した. ファジングの結果は確率的に変化するため、各手法に対して 100 回ずつ実行した結果の分布を表 4 に示す. 評価の結果、提案手法は既存手法である DIFUZZRTL [14] に比べて、短時間でバグを発生させる入力を生成できた. 提案手法は、ユーザの選択したレジスタと、その状態を決める要因となるレジスタの組み合わせからなるカバレッジ指標を用いて探索を行う. そのため、提案手法は既存手法より状態機械の状態を正確にフィードバックし、より効率的な探索を実現できた.

5.2.2 キューを内蔵するハードウェア設計

キューを内蔵するハードウェア設計を用いた実験では、キューの使用量を最大にする入力が生成できるまでの試行回数を評価した. ファジングの結果は確率的に変化するため、各手法に対して 100 回ずつ実行した結果の分布を表 5 に示す. 評価の結果、提案手法は既存手法である DIFUZZRTL [14] に比べて、短時間でキューを一杯にする入力を生成できた. DIFUZZRTL ではキューの使用量を格納するレジスタが制御レジスタとして認識されないが、提案手法ではキューの使用量を格納するレジスタを直接選択している. そのため、提案手法はキューの使用量に直接関

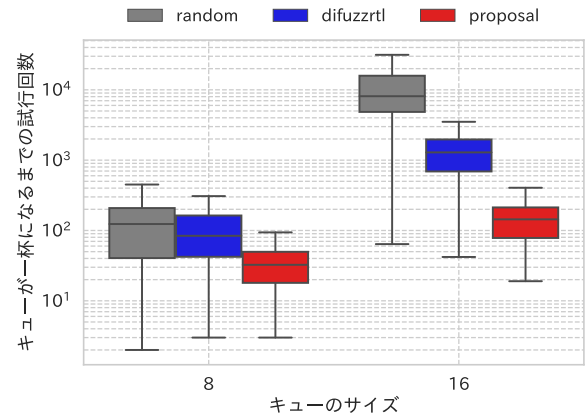


図 5: キューを内蔵するハードウェア設計に対する結果の分布

係するレジスタの情報をフィードバックし、より効率的な探索を実現できた.

5.3 実際のハードウェア設計による評価の結果

比較的規模が大きく複雑なプロセッサ設計である RSD [19] を用いた実験では、提案手法を用いて、未知のバグを発見できるか評価した. 本節の評価ではユーザの支援として、キューの使用量を格納するレジスタの選択と、キューの使用量が大きくなる方向に優先度を設定した. これは、CPU ではキューが一杯の場合の例外処理は複雑になりがちであり、バグが発生しやすいという仮定に基づいている. 評価の結果、ロード・キューとストア・キューの2つから未知のバグを発見できた. バグの内容は、キューが一杯のときに、さらにキューに要素を追加してしまうというものである. このような実験結果により、手動で作成するユニット・テストでは発見が難しいバグを提案手法により検出できることがわかった.

次に各バグが発生する入力が生成できるまでの試行回数を既存手法と比較し、提案手法の性能を評価した. ファジングの結果は確率的に変化するため、各手法に対して 10 回ずつ実行した結果の分布を図 6 に示す. 各実験は 30000 回の入力生成または入力の変異を行い、30000 回の実行にかかる時間は約 8 時間である. 図上の点が見つけた時間を表しており、バグが時間内に見つからなかったものは、最大値の 30000 に点を打っている.

ランダムに入力を生成する手法はロードキューのみで 1 回バグを発見できた. DIFUZZRTL [14] はロード・キューではバグを発見できたが、ストア・キューでは発見できなかった. 一方提案手法は、どちらに対しても大幅に短い時間で、バグを発見できている. 本評価により、提案手法は既存手法に比べて、バグを効率的に発見できることを確認した. 提案手法はユーザの支援を用いて設定したカバレッジ指標と探索方向に基づき、バグの発生しやすい状態空間を集中的にテストすることができる.

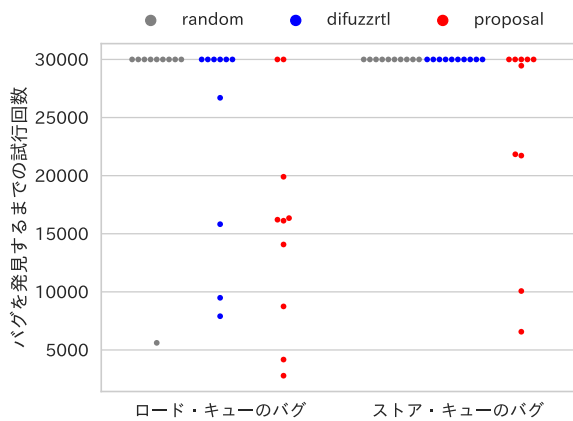


図 6: RSD に対する結果の分布

6. 関連研究

6.1 ユーザの支援を用いたソフトウェアのファジング

近年、ユーザの支援を用いたソフトウェアにおけるファジングが研究が行われている [30], [31], [32]. いくつかの研究はユーザの支援を用いて、入力生成を効率化している [30], [31]. たとえば、HaCRS [30] はテスト対象の動作に関連する可能性のある文字列のリストを提供し、ユーザがそれを選択することで入力生成を支援する. また、Dynodroid [31] はユーザの支援を用いて、アンドロイド・アプリへの入力を生成する方法を提案している. また IJON [32] では、ユーザがテスト対象のコードに対して、より詳細に探索すべき状態空間の部分に注釈を付けることを提案している. これにより、分岐などに基づくコードカバレッジより、効率的なフィードバックを返すことができるようになる. 提案手法は IJON と同じように、ユーザの支援を用いてカバレッジと探索方向を設定し、効率的な探索を実現している. 提案手法は IJON とは異なり、ハードウェア設計固有のアルゴリズムを提案していることである.

6.2 ユーザの支援を用いたハードウェアのファジング

近年、ユーザの支援を用いたハードウェアにおけるファジングとして、DirectFuzz[15] と LogicFuzzer [16] が提案されている. 本節ではそれらについて順に説明する.

6.2.1 ユーザが集中的にテストするモジュールを選択

DirectFuzz [15] はユーザが指定したモジュールを集中的にテストするための手法を提案している. DirectFuzz では、HDL で記述されたコードのモジュール階層構造を利用して、モジュール間の関係を表す有向グラフを作成する. そのグラフ上の距離に基づき、ユーザが指定したモジュールに近いモジュールのカバレッジを上げる入力を優先して変異させることで、特定のモジュールを集中的にテストする方法を提案している. このようなモジュール階層構造の距離に基づく探索方向の指標は、小さな労力で探索方向の

指標を作成できるため、小さなコストでテストを行うことができる. しかし、モジュール間の距離は必ずしも、それらの論理的な関係性の強さを表しているわけではない. そのため、モジュール階層構造の距離に基づく探索方向の指標は常に効果的ではない. これに対し、本論文ではより直接的にユーザが探索すべき箇所や条件を設定するため、より効率的に指向性のある探索を実現できる.

6.2.2 ユーザがテスト対象のハードウェア設計自体を改変

多くの既存のハードウェアにおけるファジングでは、状態空間内のバグを引き起こす状態に達するための入力を工夫して生成している [13], [14]. このような手法は簡単にテストを行うことができるが、複雑な条件でのみ到達可能な状態に至る入力を生成するためには極めて長い時間がかかる場合がある. そのような問題を解決するために、Logic Fuzzer [16] では、論理的な動作に影響を与えないようにテスト対象のハードウェア設計を変化させることで、通常の入力では到達が難しい状態に簡単に到達できるようにする方法を提案している. これにより、入力生成にかかる時間を削減できる. しかし、論理的な動作に影響を与えずに、一般にテスト対象のハードウェア設計を変化させることは難しく、多くのユーザの労力が必要である.

これに対し、提案手法は Logic Fuzzer のように多くのユーザの労力を必要とせず、効率的な入力生成を実現できる. また、Logic Fuzzer は入力空間から抽出した入力では到達不可能な状態もテストできるが、実際にどのような入力でそのバグが発生するのかわからない. 一方、本論文の提案手法では入力自体を生成するため、デバッグ時の利便性において優位と言える.

7. おわりに

これまでに多くのハードウェアにおけるファジングが提案されてきた. しかし、既存手法はハードウェアの状態空間を適切に扱うことができず、状態空間内のバグの近傍から外れたところを探索してしまっていた. これに対し、本論文ではカバレッジ指標と探索方向のそれぞれについて、ユーザの支援を用いることで状態空間を効率的に探索する手法を提案した. 本論文では提案手法を人工的に作成した様々なハードウェア設計を用いて評価した結果、既存手法と比較して状態空間を効率的に探索できることを示した. また、本論文では様々な研究において試作や評価に使われており、比較的大規模で複雑な RISC-V CPU に対して評価を行った. その結果、提案手法は小さなユーザの労力のみで、既存手法では発見が難しい、複雑なマイクロアーキテクチャの条件のみで発生する未知のバグを発見することができた.

我々は今後、ユーザの支援なしで、全自動で効率的なハードウェアのテストを実現することを目指す. そのため、ユーザの支援に相当するもの機械的に取得する必要が

ある。我々はハードウェア設計内のバグが起きやすい部分を静的解析や動的解析を用いて自動で検出し、その部分を集中的にテストすることを検討している。

謝辞 本研究の一部は、JST さきがけ JPMJPR21P5 の支援を受けたものである。

参考文献

- [1] Dessouky, G., Gens, D., Haney, P., Persyn, G., Kanuparthi, A., Khattri, H., Fung, J. M., Sadeghi, A.-R. and Rajendran, J.: HardFails: Insights into Software-Exploitable Hardware Bugs, *Proceedings of the USENIX Security Symposium (SEC)*, pp. 213–230 (2019).
- [2] Pratt, V.: Anatomy of the Pentium bug, *Proceedings of the International Joint Conference CAAP/FASE on Theory and Practice of Software Development (TAPSOFT)*, pp. 97–107 (1995).
- [3] Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T. et al.: Spectre attacks: Exploiting speculative execution, *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, pp. 1–19 (2019).
- [4] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y. and Hamburg, M.: Meltdown: Reading Kernel Memory from User Space, *Proceedings of the USENIX Security Symposium (SEC)*, pp. 973–990 (2018).
- [5] Zhang, R., Deutschbein, C., Huang, P. and Sturton, C.: End-to-End Automated Exploit Generation for Validating the Security of Processor Designs, *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pp. 815–827 (2018).
- [6] Kaivola, R., Ghughal, R., Narasimhan, N., Telfer, A., Whitemore, J., Pandav, S., Slobodová, A., Taylor, C., Frolov, V., Reeber, E. et al.: Replacing Testing with Formal Verification in Intel Core™i7 Processor Execution Engine Validation, *International Conference on Computer Aided Verification (CAV)*, pp. 414–429 (2009).
- [7] Wood, D., Gibson, G. and Katz, R.: Verifying a multi-processor cache controller using random test generation, *IEEE Design Test of Computers*, Vol. 7, No. 4, pp. 13–25 (1990).
- [8] Anderson, W.: Logical verification of the NVAX CPU chip design, *Proceedings of the International Conference on Computer Design (ICCD)*, pp. 306–309 (1992).
- [9] Chupilko, M., Kamkin, A., Kotsynyak, A., Protsenko, A., Smolov, S. and Tatarnikov, A.: Test Program Generator MicroTESK for RISC-V, *International Workshop on Microprocessor and SOC Test and Verification (MTV)*, pp. 6–11 (2018).
- [10] Squillero, G.: MicroGP—an evolutionary assembly program generator, *Genetic Programming and Evolvable Machines*, Vol. 6, No. 3, pp. 247–263 (2005).
- [11] Wagner, I., Bertacco, V. and Austin, T.: StressTest: an automatic approach to test generation via activity monitors, *Proceedings of the Annual Design Automation Conference (DAC)*, pp. 783–788 (2005).
- [12] Fine, S. and Ziv, A.: Coverage directed test generation for functional verification using bayesian networks, *Proceedings of the Annual Design Automation Conference (DAC)*, pp. 286–291 (2003).
- [13] Laeuffer, K., Koenig, J., Kim, D., Bachrach, J. and Sen, K.: RFUZZ: Coverage-directed fuzz testing of RTL on FPGAs, *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8 (2018).
- [14] Hur, J., Song, S., Kwon, D., Baek, E., Kim, J. and Lee, B.: DifuzzRTL: Differential Fuzz Testing to Find CPU Bugs, *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, pp. 1286–1303 (2021).
- [15] Canakci, S., Delshadtehrani, L., Eris, F., Taylor, M. B., Egele, M. and Joshi, A.: DirectFuzz: Automated Test Generation for RTL Designs using Directed Graybox Fuzzing, *Proceedings of the Annual Design Automation Conference (DAC)*, pp. 529–534 (2021).
- [16] Kabyllas, N., Thorn, T., Srinath, S., Xekalakis, P. and Renau, J.: Effective Processor Verification with Logic Fuzzer Enhanced Co-simulation, *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pp. 667–678 (2021).
- [17] Ghaniyoun, M., Barber, K., Zhang, Y. and Teodorescu, R.: INTROSPECTRE: A Pre-Silicon Framework for Discovery and Analysis of Transient Execution Vulnerabilities, *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pp. 874–887 (2021).
- [18] Wang, P., Krinke, J., Lu, K., Li, G. and Dodier-Lazaro, S.: How Double-Fetch Situations Turn into Double-Fetch Vulnerabilities: A Study of Double Fetches in the Linux Kernel, *Proceedings of the USENIX Security Symposium (SEC)*, pp. 1–16 (2017).
- [19] Mashimo, S., Fujita, A., Matsuo, R., Akaki, S., Fukuda, A., Koizumi, T., Kadomoto, J., Irie, H., Goshima, M., Inoue, K. et al.: An open source FPGA-optimized out-of-order RISC-V soft processor, *Proceedings of the International Conference on Field-Programmable Technology (ICFPT)*, pp. 63–71 (2019).
- [20] Liang, H., Pei, X., Jia, X., Shen, W. and Zhang, J.: Fuzzing: State of the Art, *IEEE Transactions on Reliability*, Vol. 67, No. 3, pp. 1199–1218 (2018).
- [21] Manès, V. J., Han, H., Han, C., Cha, S. K., Egele, M., Schwartz, E. J. and Woo, M.: The Art, Science, and Engineering of Fuzzing: A Survey, *IEEE Transactions on Software Engineering*, Vol. 47, No. 11, pp. 2312–2331 (2021).
- [22] Serebryany, K.: OSS-Fuzz - Google’s continuous fuzzing service for open source software, *USENIX Security Symposium* (2017).
- [23] Böhme, M., Manès, V. J. M. and Cha, S. K.: Boosting Fuzzer Efficiency: An Information Theoretic Perspective, *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pp. 678–689 (2020).
- [24] Wang, J., Duan, Y., Song, W., Yin, H. and Song, C.: Be Sensitive and Collaborative: Analyzing Impact of Coverage Metrics in Greybox Fuzzing, *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pp. 1–15 (2019).
- [25] Asanovic, K., Avizienis, R., Bachrach, J., Beamer, S., Biancolin, D., Celio, C., Cook, H., Dabbelt, D., Hauser, J., Izraelevitz, A. et al.: The rocket chip generator, *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17* (2016).
- [26] Asanovic, K., Patterson, D. A. and Celio, C.: The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor, Technical report, University of California at Berkeley Berkeley United States (2015).
- [27] Irie, H., Koizumi, T., Fukuda, A., Akaki, S., Nakae, S.,

- Bessho, Y., Shioya, R., Notsu, T., Yoda, K., Ishihara, T. and Sakai, S.: STRAIGHT: Hazardless Processor Architecture Without Register Renaming, *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pp. 121–133 (2018).
- [28] Mitsuno, S., Kadomoto, J., Koizumi, T., Shioya, R., Irie, H. and Sakai, S.: A High-Performance Out-of-Order Soft Processor Without Register Renaming, *Proceedings of the International Conference on Field-Programmable Logic and Applications (FPL)*, pp. 73–78 (2020).
- [29] Jiang, Z., Yang, K., Ma, Y., Fisher, N., Audsley, N. and Dong, Z.: I/O-GUARD: Hardware/Software Co-Design for I/O Virtualization with Guaranteed Real-time Performance, *Proceedings of the Annual Design Automation Conference (DAC)*, pp. 1159–1164 (2021).
- [30] Shoshitaishvili, Y., Weissbacher, M., Dresel, L., Salls, C., Wang, R., Kruegel, C. and Vigna, G.: Rise of the HaCRS: Augmenting Autonomous Cyber Reasoning Systems with Human Assistance, *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.
- [31] Machiry, A., Tahiliani, R. and Naik, M.: Dynodroid: An Input Generation System for Android Apps, *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pp. 224–234 (2013).
- [32] Aschermann, C., Schumilo, S., Abbasi, A. and Holz, T.: Ijon: Exploring Deep State Spaces via Fuzzing, *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, pp. 1597–1612 (2020).