

urandom

Computer Security Magazine

vol.12



Quantum Covert Lottery with Cheat Detection
yyu

代理フィードバックを用いたCPUファジングの試み
op

令和最新版urandom出版システム
mayth

目次

Quantum Covert Lottery with Cheat Detection	2
1 はじめに	2
2 秘密の希望に基づく「先手・後手」決定ゲーム	2
3 量子コンピュータの基礎	3
4 量子ゲートテレポーテーション	7
5 ボブの不正検出付き Quantum Covert Lottery	10
6まとめ	15
代理フィードバックを用いた CPU ファジングの試み	16
1 はじめに	16
2 Zenbleed とは	16
3 実験環境	17
4 実験設計	19
5 実装	19
6 実験結果	21
7 考察	23
8 おわりに	24
令和最新版 urandom 出版システム	26
1 はじめに	26
2 システム構成	26
3 改修の内容	27
4 なにもしてないのにこわれました	33
5 最後に	36

Quantum Covert Lottery with Cheat Detection

1 はじめに

“Covert Lottery”とは2021年に発表された論文“Card-Based Covert Lottery”[1]で発表されたガチャの1つです。これは2人プレイヤーの秘密の希望(1bit)を入力して、もし2人の希望が衝突しないのであれば希望通りにし、そうでないならランダムな結果を出力するというプロトコルです。

2人によるこのプロトコルを量子コンピュータで実装したものをZenn.devの記事[2]にしてから、C100で発行したurandom vol.9ではCovert Lotteryを多人数に拡張するためにCovert Lotteryとグローバーのアルゴリズムについて考えました。今回の記事では多人数拡張ではなくて、もともとの2人プレイヤーにいったん戻り不正検出(*cheat detection*)について考えていきます。[2]のプロトコルではプレイヤーの間で2つの量子ビットをやりとりしながら量子計算を行いますが、片方のプレイヤーのみが最終的なプロトコルの出力に直結する結果を得るため、その値を改竄されると残ったプレイヤーにはそれを見抜く手段がありませんでした。

この記事では最初におさらいとしてCovert Lotteryの性質を解説したあと、量子計算の基礎的なところを説明しながら不正検出付きの量子Covert Lotteryについて解説します。

なお、この記事は第56回情報科学若手の会^{*1}で発表した内容を基にしています。

2 秘密の希望に基づく「先手・後手」決定ゲーム

まずは2人のCovert Lotteryについておさらいします。すでにurandom vol.9などや筆者のZenn.devの記事[2]を読んでいる場合、この節は飛ばしても構いません。

この節では元論文で説明されている2人のCovert Lottery(先手・後手を秘密に決定するゲーム)について説明します。今2人のプレイヤーとしてアリスとボブがいるものとし

^{*1} <https://wakate.org/2023/08/04/56th-general/>

- 1 量子ビット目の測定結果が $|0\rangle_2$ 2 量子ビット目にはテレポーテーションした H ゲートのみが作用して $H|+\rangle_2 = |0\rangle_2$ となる
- 1 量子ビット目の測定結果が $|1\rangle_2$ 2 量子ビット目には CZ ゲートによる Z ゲートが作用、その後テレポーテーションした H ゲートも作用するため、結果として $H(Z|+\rangle_2) = H(|-\rangle_2) = |1\rangle_2$ となる

よって最終的な結果として $|00\rangle$ と $|11\rangle$ がそれぞれ確率 $\frac{1}{2}$ で観測されます。

また、 CZ ゲートは今回紹介した H ゲートの他に、 Z, S ゲートをテレポーテーションさせます。

4.2 適応的な量子ゲート操作

図 6 の回路について再び考えます。1 量子ビット目の結果によって 2 量子ビット目がどのような影響を受けるのかを量子回路として図 8・図 9 に示します。

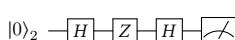


図 8: 測定結果が $|0\rangle$ の場合の回路

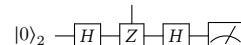


図 9: 測定結果が $|1\rangle$ の場合の回路

この回路は H ゲートが 2 量子ビット目へテレポーテーションする作用と、1 量子ビット目の測定結果に応じて 2 量子ビット目に CZ ゲートが作用するかどうかの 2 つが同時に発生し、1 量子ビット目の結果に応じて偶発的に CZ ゲートが作用するかどうかが決まります。言いかえると量子ゲートテレポーテーションに用いる CZ ゲートによって、1 量子ビット目の結果に 2 量子ビット目が依存してしまいます。

ここで古典コンピュータの `if` のように、1 量子ビット目の測定の結果が $|0\rangle$ なのか $|1\rangle$ のかに応じて、適応的に 2 量子ビット目に X ゲートを作用させるような図 10 の回路を考えます。

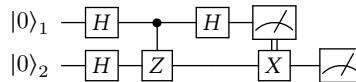


図 10: 1 量子ビット目の測定結果に応じて X ゲートを作用させる回路

この回路では 1 量子ビット目が $|0\rangle$ であれば特に何もせず、一方で $|1\rangle$ であれば 2 量子ビット目に X ゲートを作用させます。

このようにすることで、図 6 の回路は 1 量子ビット目が $|1\rangle$ であれば 2 量子ビット目が $|1\rangle$ が測定されていたため、このとき 2 量子ビット目に X ゲートを作用させ $X|1\rangle = |0\rangle$ となります。1 量子ビット目の測定の結果として $|0\rangle$ が観測された場合、2 量子ビット目は

代理フィードバックを用いた CPU ファジングの試み

1 はじめに

Spectre や Meltdown を初めとした CPU の脆弱性がコンピューターセキュリティにおける懸案事項となる中、機械語実行器としての CPU に対してファジングを適用し脆弱性を探索する試みが一部でなされている。一般的なファジングではコードカバレッジ等によるフィードバックが性能の鍵となる。しかしながら、市販の CPU では命令セットアーキテクチャの範囲を超えて詳細な内部状態をつぶさに観察できないため、有意なフィードバックを得るのが難しいという問題がある。この問題に対し、CPU エミュレーター等に機械語命令列を入力し、そのエミュレーターのコードカバレッジを「代理」(Proxy) のフィードバックと見なしてファジングの性能を向上させるアイデアが提案されている^{*1}。この記事ではこのアイデアを参考に、オープンソースの CPU エミュレーターである Unicorn^{*2} のコードカバレッジを利用して CPU のファジングを試みる。ファジングの対象は Zenbleed の実証コード中にある投機的実行機能を訓練する部分のコード片 (*training code*) とし、より高確率で Zenbleed を引き起こす *training code* の発見を目標とする。

2 Zenbleed とは

Zenbleed は、AMD 社の Zen 2 プロセッサーにおいて、あるプロセスから別プロセスのレジスターの一部を読み取れる脆弱性である。この脆弱性は VZEROUPPER 命令の実行時に分岐予測ミス（投機的実行のロールバック）が発生した時、確率的に発見すると考えられている。つまり、Zenbleed の発現確率は分岐予測ミスの発生確率に依存すると考えられている。

分岐予測ミスを利用する攻撃では、攻撃を行う前に予め *training code* と呼ばれる短い

^{*1} SiliFuzz: Fuzzing CPUs by proxy <https://arxiv.org/abs/2110.11519>

^{*2} Unicorn – The Ultimate CPU emulator <https://www.unicorn-engine.org/>

命令列を実行して分岐予測器を訓練することで、分岐予測ミスの発生率を上げる実装が一般的である。例として発見者が公表している Zenbleed の実証コード^{*3}を抜粋^{*4}して次に示す。

```

1  zen2_leak_train_mm0:
2      vzeroall
3      xor      rax,  rax
4      movd    mm0,  rax
5      align    64
6 .restart:
7      mov      rcx,  90
8 .again:
9      dec      rcx
10     cvtpi2pd xmm4,  mm0
11     cvtpi2pd xmm3,  mm0
12     cvtpi2pd xmm2,  mm0
13     cvtpi2pd xmm1,  mm0
14     cvtpi2pd xmm0,  mm0
15     vmovdqa ymm0,  ymm0
16     js      .overzero
17     vzeroupper
18 .overzero:
19     jns      .again
20     vptest   ymm0,  ymm0
21     vptest   ymm0,  ymm0
22     jz      .restart
23     lfence
24     vmovdqu [rdi],  ymm0
25     ret
26     hlt

```

このコードにおいては `.restart` ラベルから `.overzero` ラベルの間の命令が training code と言える。このコードでは `.again` から始まるループを 91 回実行するが、Zenbleed を発見させる VZEROUPPER 命令は最後の 91 回目にのみ実行される。90 回目までのループで CPU の分岐予測器は `js .overzero` 命令の分岐結果を学習するため、91 回目では分岐予測を誤りがちになる。また、`cvtpi2pd xmm4, mm0` から `vmovdqa ymm0, ymm0` までの 6 命令も一連の学習に寄与していると思われる。

この記事では training code のうち `cvtpi2pd xmm4, mm0` から `vmovdqa ymm0, ymm0` までの区間の命令列をファジングで探索し、オリジナルの実証コードよりも高確率で Zenbleed が発生する training code の発見を目標とする。

3 実験環境

記事中で使用した実験環境は次の通り。

CPU

AMD Ryzen 5 3600 6-Core Processor (6C12T)

^{*3} <https://github.com/google/security-research/blob/93b3849/pocs/cpus/zenbleed/zenleak.asm#L122>

^{*4} 紙面の関係上、コード中のコメントは筆者が削除した。

令和最新版 urandom 出版システム

1 はじめに

urandom vol.10^{*1}にて『urandom 出版システム 2022 Edition』を執筆して 1 年が経ちました。C99（2021 年 12 月）の時点では TeXLive の Docker イメージは x86_64 向けにしか存在せず、メンバーのマシン上での再現性あるビルドができなくなってしまい、CI システムの必要性が高まっていました。ところがその次、2022 年 8 月の C100 の時点でも CI システムは動いていませんでした。このように完全に腐りきっていた CI システムをどうにかして動くところまで持っていく、というのが前回の記事でした。

出版システムに関しては毎度「あれはこうした方がよかった」「ここはこうすると早くできそう」などと入稿直後には話しているものの、そんなことはコミケに参加して薄い本を目の前にすれば吹き飛んでおり、結局次にシステムに目を向けるのは締切 1 ヶ月前になるわけです。そんな時期ではシステムを壊した場合に取り返しが付かないでの今からシステムを弄っている場合ではない、という状況になっているのが常でした。

今回はいよいよ重い腰を上げていくらかの改修を試みたので、この記事ではその報告をします。

2 システム構成

前回のシステム構成をざっと述べると次の通りでした。

- ソースコード管理: GitBucket
- ビルドシステム: Jenkins
- サーバー: さくらの VPS 2G プラン（東京リージョン）

GitBucket と Jenkins はさくらの VPS で借りたサーバーで Docker を用いて動作させていました。

^{*1} <https://urandom.team/books/urandom-vol10/> C101（2021 年 12 月）にて頒布。

3 改修の内容

改修の作業中にはトラブルもありましたが、先にどのような改修を、どのように行ったのかを説明します。

今回の改修では、まず GitBucket を GitHub に移行しました。また、Jenkins のビルドノードを VPS からメンバー宅内のサーバーに切り替えました。一旦しばらくはこの状態で運用し、最終的にはビルドシステムも GitHub Actions に移しました。

3.1 GitHub への移行

前回時点でも GitHub への移行は視野には入れていました。原稿を Git で管理するようにした当初はプライベートリポジトリの数に制限がかけられており、それが GitBucket を使っていた理由でした。現在は（というか前回記事の時点でも既に）プライベートリポジトリ数に制限はなく、特に使わない理由もなくなっていたので、GitHub に移すことにしました。

リポジトリそのものはどこにホストしているが Git のものに違いありません。実のところ C103 に向けた作業の開始時点では GitHub に移行するつもりはなく、いつも通り GitBucket にリポジトリを作っていました。そのため、手元にあるリポジトリが GitBucket を向いている状態から作業を開始します。今回は次のように `git remote set-url` で origin を GitHub にして `push` するという手段を取りましたが、後になって「リポジトリを移行するには bare で `clone` してきてから `git push --mirror` をするんだ」とする記事を見つけました²。移行とミラーリングでは意味が異なるような気もしますが、動くならなんでもよいでしょう。

```
$ git remote set-url origin https://github.com/hoge/piyo.git  
$ git push
```

さて、リポジトリをホストする先を切り替えたところで、連携するシステムの方も修正しなくてはなりません。といっても連携先は Jenkins しかありませんから、Jenkins 内での変更箇所を洗っていきます。

まず Jenkins から GitBucket にアクセスする際の権限情報を確認します。Jenkins の Web UI にアクセスし、Jenkins の管理 → Credentials と辿って、対象の credential の詳細を表示します。すると、そこに Usage という項目があり、ここにその権限を使用しているジョブの一覧が出てきます。次のスクリーンショットは既に移行を済ませた後に取ったので何も表示されていませんが、ここにジョブがあれば、そのジョブは GitBucket の credential を使っているため移行が必要です。

² <https://docs.github.com/ja/repositories/creating-and-managing-repositories/duplicating-a-repository>

urandom vol.12

発行者	urandom
表紙デザイン	秋弦めい
発行日	2023年12月31日
バージョン	1.00
コミット ID	b81e0d1
連絡先	https://urandom.team/
印刷所	サンライズ

urandom
•ω•

presented by urandom

Comic Market 103 (Dec. 31st, 2023)