PoCoFuzz : Power Consumption aware Fuzzing

SCIS 2019 2019 Symposium on
Cryptography and Information Security
Shiga, Japan, Jan. 22 - 25, 2019
The Institute of Electronics,
Information and Communication Engineers

PoCoFuzz: 消費電力を考慮したファジング
PoCoFuzz: Power Consumption aware Fuzzing

水野　慎太郎 *　　西垣　正勝 *　　大木　哲史 *
Shintaro Mizuno　　Masakatsu Nishigaki　　Tetsushi Ohki

あらまし 近年，ノートパソコン，モバイル機器，IoT 機器など，様々な機器がバッテリーを使用している．バッテリー消費はエンドユーザーの利便性に深く関わる要素であるが，悪意ある攻撃を想定した場合，その脅威はエンドユーザーの利便性低下にとどまらず，大量の電力消費を利用した電源供給設備への攻撃にまで脅威が及ぶ可能性がある．大量な消費電力が発生する原因の一つとして，消費電力の大きいプログラムの動作が挙げられる．中でも，入力に依存するプログラムの場合，入力を変化させることで消費電力を大きくすることが可能となる．本稿では，消費電力の大きい要因を調査し，要因となり得る入力をファジングによって自動的に発見することで，消費電力の大きい入力を自動的に発見する PoCoFuzz を提案する．ファザーである「American Fuzzy Lop」を用いて PoCoFuzz を実装・評価することで，提案手法の有効性を示す．結果として，検証プログラムに対して，初期に生成された入力と比較して，約 17.4 倍消費電力が大きいと推定される入力の生成に成功した．

キーワード 消費電力，ファジング，動的解析

1　はじめに

プログラムは，日々大規模化・高度化しており，開発者が注意して開発を行った場合であっても，予期せぬ脆弱性が含まれる可能性がある．含まれた脆弱性が深刻なものであった場合，攻撃者に悪用される可能性もあるため，早期に除去する必要がある．しかし，脆弱性の種類は多岐に渡り，事実，ゼロデイの脆弱性は近年においても多数報告されている．このことからも，開発時に全ての脆弱性を取り除くことには限界があると言える．したがって，実装されたプログラム中に潜む，意図せぬ脆弱性を発見する手法を検討する必要があり，これを実効的に行う手法としてファジングが提案されている．近年，多くのファジングに関する研究により，効率的に未遷移のコードを探索する手法や，脆弱性の発生する入力を効率的に生成する手法が数多く提案されている．さらに，ファジングの技術を応用し，脆弱性だけではなく，プログラム実行時間の増加など，望ましくないイベントを発見するための研究も行われている．IoT 機器をはじめとした，多種多様な機器・サービスが混在する環境下では，望ましくないイベントの種類もまた，環境やサービスごとに多角化していく．このことから，実装されたプログラム中に潜む多種多様なイベントを発見する方法として，ファジングを用いることは，有意義である．

本稿では，これらの多種多様なイベントのうち，機器の電力消費を増大させるイベントに着目し，ファジングを応用した解決手法を提案する．機器の電力消費は，その増大要因が多様であることから，原因の特定が困難であるとともに，さらにその影響範囲も多岐にわたる．たとえば，バッテリー駆動する電子機器の場合，バッテリーの駆動時間を長く保つため，プログラムの開発者は大量の消費電力を発生させないよう，注意してプログラムを設計する必要がある．また，IoT 機器では，複数の機器による，同時期の周波数の上昇・低下やシャットダウンといった攻撃によって，電源供給設備に対して意図的な停電を引き起こすことの可能性が指摘されている [9].

大量の消費電力発生を防ぐために，デバイスの低消費電力化やソフトウェアの消費電力の見える化など多くの手法が検討されてきた．一方で，消費電力が増大する原因を考えると，トランジスタやメモリといったハードウェア的要因，大規模データの処理やプロセスの状態遷移といったソフトウェア的要因など，様々な要因が想定される．特に，これらの要因のうち，外部からの入力，たとえば Web アプリケーションにおけるユーザ入力や IoT センサにおけるセンサへの入力，に依存する処理は，その結果が入力内容に依存する．この場合，入力と電力消費の要因との関係が複雑となることから，消費電力の推定が非常に困難となる．このため，開発時に気づきにくい

* 静岡大学, 浜松市中区城北 3 丁目 5-1, Shizuoka University, 3-5-1 Johoku, Naka-ku, Hamamatsu City, Shizuoka, Japan

異常な入力等を利用した攻撃が可能となり，IoT 機器を電力供給不足へ追い込む攻撃や，バッテリーを浪費させる攻撃など，様々な用途で悪用される危険が生じる．

本稿では，従来のファジングでは不可能であった，プログラムの脆弱性以外の情報である消費電力の増大を自動的に検知するために，メモリアクセスやディスクアクセスといった電力消費に関連する処理を多く含む実行経路 (以下，消費電力経路と呼ぶ) ほどプログラム実行時の消費電力が大きいという仮説の下に，動的解析手法とファジングを組み合わせて得られた情報を用いて，消費電力が多いと推定される入力値を効率的に探索する手法を提案する．本論文ではこの手法を PoCoFuzz と名付け，本手法の有効性を評価する．まず，本手法が想定する，電力消費に関連する処理（これらを消費電力要因と呼ぶ）について，各要因と消費電力との関係を明らかにし，これを消費電力モデルとしてモデル化する．次にこのモデルとファジングを組み合わせることで，電力消費を増大させる入力を効率に特定することが可能となる．評価の結果，初期の入力と比較して約 17.4 倍消費電力が大きいと推定される入力の生成に成功した．

本論文の貢献は，次のようにまとめられる．

- 消費電力要因と消費電力の関係を明らかにするために，消費電力要因が発生させる消費電力を調査し，消費電力モデルを作成する．
- 消費電力モデルの消費電力要因を追跡する動的解析と，ファジングを組み合わせることで，消費電力が多いと推定される入力値を探索する PoCoFuzz を提案し，その有効性を評価する．

2 関連研究

2.1 ファジング

ファジングは，プログラムに対して大量に自動で生成した入力データ (標準入力，ファイルなど) を送り，その応答・挙動によって脆弱性検知を行うテスト手法の一種である．ファジングは，Miller ら [6] によって提唱された概念であるが，当時はランダムな文字列を大量に生成し，それをプログラムへの入力データとすることで，クラッシュやハングを判定するのみであった．しかし，近年では，AFL[4] や libFuzzer[1] といった，ファジングを行うソフトウェアであるファザーによって，クラッシュやハングだけでなく，異常終了や任意コード実行といった多くの脆弱性が発見されている．また，ファジングにおける入力データの自動生成手法としては，AFL のような遺伝的アルゴリズムを使用した手法，Driller[10] のようなシンボリック実行を組み合わせた手法，VUzzer[8] のような静的解析と動的解析を組み合わせる手法など，

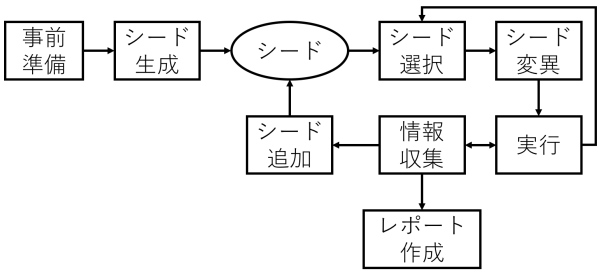

図 1: ファジングの主な流れ．ファザーによっては，いくつかの手順を省略している．

様々な手法が提案されている．従来のファジングはそのほとんどが異常終了や任意コード実行という脆弱性を引き起こす入力の発見を目的としたものであったが，近年では，たとえば，実行速度が極端に低下するような入力をファジングにより発見する SlowFuzz[7] など，本来意図しない動作そのものを発見する手法へとその応用範囲が広がりつつある．

ファジングは一般的に，図 1 に示したような手順で実行される．まず事前準備を行うことで，変異させる入力データとなるシードの生成を行い，シードとして保持する．実行時には，保持されたシードから任意のシードを選択し，シードの一部を変異させ，解析対象プログラムへの入力とする．ファザーは解析対象プログラムの実行中に，基本ブロック間の遷移情報や特定の動作といった情報を収集する．この時，意図しない動作が発生すればレポートの作成を行う．また未遷移の実行経路に遷移した入力などを興味深い入力と判定し，その入力をシードに追加する．この動作を繰り返し行うことで，シードにはより興味深い入力が保持され，解析対象プログラムに対して洗練された入力を与えることが可能となり，意図しない動作の発生する入力データを生成できる可能性が高まる．

2.2 消費電力の推定

消費電力の推定にあたっては，CPU やメモリといった構成要素の動作と消費電力の関係を数学的にモデル化する（以下，電力モデルと呼ぶ）ことで，実際の動作時における消費電力の推定を行う手法が存在する．Hayri ら [2] は，CPU，メモリ，ディスクの 3 要素の電力モデルについて，様々な先行研究の電力モデルを提示し，それらを元に，消費電力を見積もるツールを紹介した．また，同時に最適化されていないソースコードは，最適化されたソースコードよりも消費電力が高いことを示した．神山ら [11] は，ディスプレイや GPS といった計 8 要素の電力モデルを用意し，アプリケーションごとのログ収集と組み合わせることで，アプリケーションごとの消費電力を推定している．

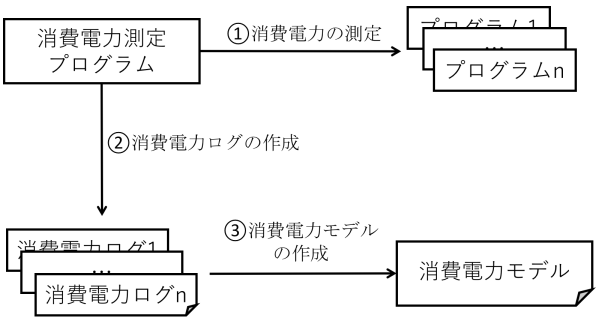

図 2: 消費電力モデルの作成手順

Hayri ら，神山らの手法では，要素ごとの電力モデルを用意することで，消費電力の推定を行った．これに対して，オープンソースである PowerTOP[3] は，GPU の使用回数といった要素の電力モデルに加えて，Xorg 系のプロセス起動についても消費電力の推定を行う．ftrace と呼ばれるトレース機構を使用することで，プロセスごとのコンテキストスイッチや割り込み要求といったイベントを追跡することで，プロセスごとの消費電力を推定している．

上記 3 つの手法においては，入力に依存したプログラムに対して，入力値によって消費電力が変化する，という点が考慮されていない．それに対して，本稿で提案する消費電力を考慮したファジング手法では，消費電力要因を用いて消費電力経路を効率的に探索することで，入力値の変化による消費電力の増大の発見を可能とする．

3 提案手法

3.1 提案手法概要

PoCoFuzz は，Web サイトの検索欄やセンサ情報などといった外部からの入力によって，大量の消費電力を発生させることが可能であるという仮定の下，大量の消費電力を発生させる入力をファジングによって自動的に発見する手法である．大量の消費電力を発生させる入力を発見するため，まず消費電力モデルの作成を行う．次に，消費電力モデルを元に，消費電力経路をファジングによって発見する．

3.2 消費電力の多い入力の発見手法

3.2.1 消費電力モデルの作成

大量の消費電力を発生させる入力を発見するにあたり，まず消費電力モデルの作成を行う．消費電力モデルの作成を行う手順を図 2 に示す。

消費電力モデルの作成には，消費電力要因と考えられる動作を行うプログラム群が必要となる．消費電力に大きく関係する要因として，メモリアクセス，ディスクアクセス，通信など，様々な要因が挙げられる．これらの

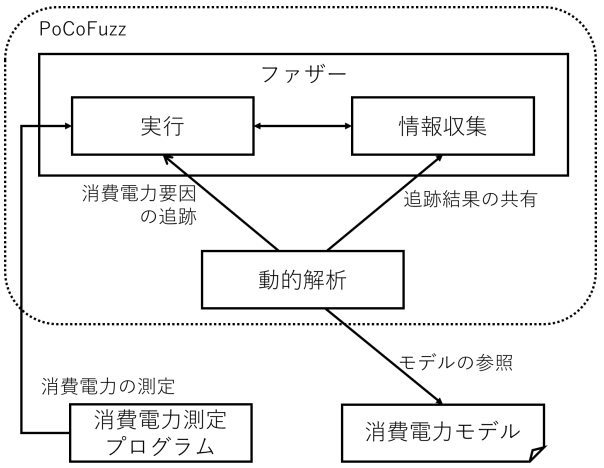

図 3: 動的解析とファジングを用いた消費電力要因の追跡

プログラムを一定時間動作させた結果を消費電力測定プログラムで計測することで，各要因が消費電力に与える影響を推定する．ここで，測定を行うにあたっては，プログラム内における単一の消費電力要因による消費電力への影響のみを測定するために，メモリアクセスのみを行うプログラム等，単一要因による負荷を定期的に与えるプログラムを作成し，実行する．このようにして推定した消費電力要因および測定した消費電力を元に，消費電力モデルを作成する．消費電力要因を $f = \{f_1, f_2, \cdots, f_n\}$，時刻 t における f_i の消費電力を $p^t_{f_i}$ としたとき，消費電力モデル $M = \{M_{f_1}, M_{f_2}, \cdots, M_{f_n}\}$ の各要素 M_{f_i} は，消費電力要因 f_i と，f_i の単位時間あたりの消費電力の期待値 $\bar{p}_{f_i}$ の組 $\{f_i, \bar{p}_{f_i}\}$ によって構成される．$\bar{p}_{f_i}$ は，式 (1) によって求められる．

$$\bar{p}_{fi} = \mathop{\mathbb{E}}_{t} p^t_{fi} \tag{1}$$

3.2.2 消費電力要因の追跡

次に，消費電力モデルを参照しつつ、解析対象プログラムにおける消費電力要因の追跡を行う．消費電力要因の追跡を行う際の，各構成要素および構成要素間の関係を図 3 に示す．

各構成要素は，以下の役割を持つ．

- 消費電力測定プログラム: ファザー内で実行されている解析対象プログラムの消費電力を測定する．
- 消費電力モデル: 動的解析による消費電力要因の追跡の際に使用される．
- 動的解析: ファザー内で実行されている解析対象プログラムの実行経路上に存在する，消費電力モデルに記述されている消費電力要因を追跡する．動的解析による追跡結果は，ファザー内の情報収集プロセスと共有を行うことで，以降の入力データ生成に役立てる．

アルゴリズム 1 PoCoFuzz の主要なアルゴリズム．解析対象プログラム P に対して，シード S を用いて n 回ファジングを行い，消費電力経路を経由する入力の集合 *power_units* を返す

```
1:  procedure PoCoFuzz(P, S, n)
2:      power_units = ∅
3:      cnt = 0
4:      while cnt < n and S ≠ ∅ do
5:          in =SelectSeed(S)
6:          in′ =MutateInput(in)
7:          power_data =Run(P, in′)
8:          if IsInteresting(power_data) then
9:              S ∪ = in′
10:             power_units∪ = in′
11:         end if
12:         cnt = cnt + 1
13:     end while
14:     return power_units
15: end procedure
```

- ファザー: 図 1 のように，大量の入力データを生成しながら，解析対象プログラムを実行する．解析対象プログラムの実行時，動的解析による消費電力要因の追跡，消費電力測定プログラムによる消費電力の測定を行う．またファザー内で行われる通常の情報収集に加えて，動的解析の追跡結果を利用することで，消費電力経路を経由する入力を発見する．

各構成要素を使用した PoCoFuzz の主要なアルゴリズムを，アルゴリズム 1 に示す．PoCoFuzz では，任意のシードを選択 (5 行目) し，シードを変異させる (6 行目)．変異したシードを解析対象プログラムへの入力とし，解析対象プログラムを実行する (7 行目)．解析対象プログラムの実行中，動的解析により，消費電力モデル M を参照しながら，解析対象プログラムが実行した消費電力要因の集合 $PF = \{PF_1, PF_2, \cdots, PF_x\}$ を追跡する．追跡結果から消費電力経路の実行の有無を判定する値として，式 (2) を計算し，共有メモリなどを介して，ファザーの情報収集プロセスと共有する．

$$\sum_{PF_j \in PF} \sum_{f_i \in f} \bar{p}_{[PF_j = f_i]} \tag{2}$$

同時に，消費電力測定プログラムによって，解析対象プログラムの消費電力を測定する．追跡した情報を利用して，入力が消費電力経路を実行していると思われる場合には，その入力をシードに追加する (8–11 行目)．

4 実装

4.1 消費電力要因の動作を行うプログラムの作成

消費電力要因の動作を行うプログラムでは，消費電力測定プログラムとして PowerTOP を使用する．全てのプログラムは，消費電力要因の動作を行い，20 ミリ秒ごとに usleep() を行うように作成した．20 ミリ秒ごとの usleep() を行う必要性については，6.1 節で議論している．消費電力要因の動作を行うプログラムとして，メモリアクセス，ディスクアクセス，メモリ確保，通信を用意した．また，全てのプログラムで実行されている usleep() についても用意した．メモリ確保の動作を行うプログラム例を，図 4 に示す．

```
int main(void){
    while(1){
        char *ret;
        int fd;

        ret = mmap(NULL, 4096, PROT_READ|
            PROT_WRITE, MAP_PRIVATE|
            MAP_ANONYMOUS, -1, 0);
        munmap(ret, 4096);
        usleep(20000);
    }
}
```

図 4: mmap() によるメモリ確保を行うプログラム

4.2 PoCoFuzz の実装

PoCoFuzz のプロトタイプの実装にあたり，ファザーとして AFL，動的解析として ptrace を使用したプログラムを使用した．消費電力の測定に関して，3.2.2 項で示した提案手法では，ファザー内で実行されている解析対象プログラムの消費電力を測定していた．それに対して，プロトタイプでは，ファジングによって生成された消費電力経路を経由する入力と，解析対象プログラムを一定時間ごとに動作するプログラムを用意しておき，生成された入力をプログラムへの入力とした状態で，PowerTOP による消費電力の推定を行う．

ptrace を用いた動的解析では，メモリ確保，ディスクアクセス，通信をシステムコールによって追跡する．例えば，メモリ確保であれば brk システムコール，または mmap システムコールを追跡する．これらのシステムコールを追跡することで，どの消費電力要因の動作を行っているか判別する．4.1 節のうち，メモリアクセスに関して追跡していない理由については，5.1 節で述べる．AFL では，ptrace を用いた動的解析および AFL によって得られた情報を元に，消費電力経路を経由する入力を，優先的にシードへ追加する．

5 実験・検証

ここでは，消費電力モデルに追加する消費電力要因の選定，バイト数の変化による消費電力の変化に関する調査，PoCoFuzz を使用した際の消費電力の増加に関する検証を

行う．全ての実験・検証は，インテル (R) Core i7–8750H プロセッサと 8GB のメモリを搭載した，Ubuntu16.04 上で行った．また，カーネルはバージョン 4.4.0，gcc はバージョン 5.4.0，PowerTOP はバージョン 2.8，AFL はバージョン 2.52b を使用した．

5.1 消費電力モデル作成に関する実験

4.1 節で作成した消費電力要因の動作を行うプログラムを実行し，それぞれの消費電力を 30 分間記録した．記録した消費電力は，0 mW といった極端に小さな値を取り除き，式 (1) を計算することで，消費電力モデルに追加する際の消費電力値とした．消費電力要因ごとの，動作中に呼ばれるシステムコールおよび消費電力を表 1 にまとめた．

配列参照によるメモリアクセスと，usleep() によるスリープの消費電力要因について，消費電力がほぼ等しい．usleep() によるスリープは，全てのプログラムで実行されているため，この消費電力要因とほぼ等しい消費電力要因については，消費電力モデルから除外すべきである．そのため，今回はメモリアクセスに関する追跡は行わないものとし，消費電力モデルへの追加を行わなかった．

その他の消費電力要因は，手法による差異はあるものの，ディスクアクセス，メモリ確保，通信ごとにある程度まとまった消費電力となっている．消費電力は，通信，ディスクアクセス，メモリ確保の順に大きいと推定されている．実験・検証を行った電子機器において，最も電力を消費する消費電力要因は，サーバによる接続であった．メモリアクセスを除いた全ての消費電力要因は，usleep() によるスリープと比較して，多くの電力を消費するため，消費電力モデルに加えることとした．

5.2 バイト数による消費電力の違いに関する実験

4.1 節で作成した消費電力要因の動作を行うプログラムにおいて，1 回の read()/write() で行われる読み書きのバイト数，1 回の mmap()/sbrk() で行われるメモリ確保のバイト数は，全てのプログラムで 4096 バイトであった．ここでは，read() によるディスクアクセスと write() によるディスクアクセスについて，バイト数を変化させた場合，消費電力の変化が発生するか調査を行った．

read()/write() のバイト数について，4，16，64，256，1024，4096 バイトに変化させたプログラムを用意し，5.1 節と同様に消費電力を記録し，式 (1) にしたがって計算を行った．バイト数を変化させた時の read()/write() の消費電力について，図 5 にまとめた．

消費電力に関して，16 バイト時の消費電力をのぞいて，バイト数の変化による大きな消費電力差は存在しない．また，バイト数と消費電力が比例しているわけではなく，ばらつきが見られる．以上のことから，今回はバイト数による消費電力の変化はないものと判断し，消費電力要因ごとの消費電力のみ消費電力モデルに追加した．なお 16 バイト時の消費電力が大きい点については，現在調査中である．

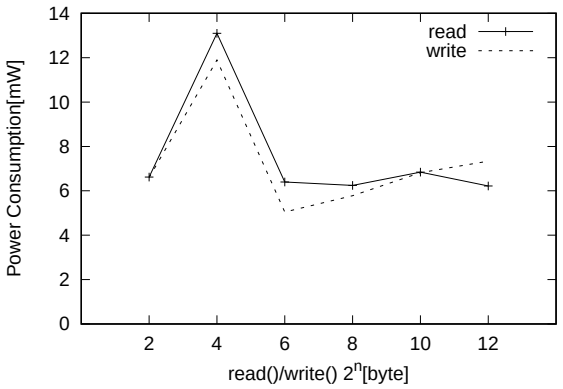

図 5: バイト数ごとの read() と write() の消費電力

5.3 PoCoFuzz の検証

PoCoFuzz の検証にあたり，検証プログラムを用意した．検証プログラムでは，標準入力から 8 バイトの入力を取得し，その入力によって nanosleep システムコールを除いた消費電力要因の動作を複数回行うものである．プログラムの概要を，アルゴリズム 2 に示す．

検証プログラムでは，標準入力 8 バイトを受け取り，最初の 2 バイトが消費電力要因の種類の選択に使用され (6–26 行目)，後に続く 4 バイトが消費電力要因の実施回数の計算に使用される (2–5 行目)．なお，残りの 2 バイトについては，使用しないダミーとなっている．

検証プログラムに対して，初期のシードを 0x68656c6c6f とした状態で，PoCoFuzz を使用したファジングを実行した．1 時間 30 分ほど実行した結果，シードには計 103 つの値が追加され，最後に追加されたシードの値は 0x01007f808080 であった．

初期のシードである 0x68656c6c6f は，nanosleep のシステムコールを除いた mmap() によるメモリ確保を 1 回実行するものであった．それに対し，最後に追加されたシードである 0x01007f808080 は，nanosleep のシステムコールを除いたクライアントの write() による接続を 511 回実行するものである．クライアントの write() による通信は，検証プログラムに含まれている，複数回実行する消費電力要因の中で，最も大きい消費電力値の消費電力要因である．また，511 回については，消費電力要因の実行回数における最大値である．以上のことから，PoCoFuzz は検証プログラムに対して，最も大きい消費電力経路を経由する入力を生成することができた．

生成されたシードを使用して，消費電力の記録を行っ

表 1: 消費電力要因ごとの消費電力

消費電力要因	1 回のループで呼ばれるシステムコール	消費電力 [mW]
read() によるディスクアセス	read，nanosleep	5.44177
write() によるディスクアクセス	write，nanosleep	6.37688
mmap() によるディスクアクセス	mmap，munmap，nanosleep	6.67455
brk() によるメモリ確保	sbrk，brk，nanosleep	5.03481
mmap() によるメモリ確保	mmap，munmap，nanosleep	5.04911
クライアントの read() による通信	read，nanosleep	9.54255
クライアントの write() による通信	write，nanosleep	10.26677
サーバの read() による通信	read，nanosleep	9.94977
サーバの write() による通信	write，nanosleep	9.66077
クライアントによる接続	socket，stat，connect，close，nanosleep	11.54800
サーバによる接続	socket，setsockopt，bind，listen，accept，close，nanosleep	12.55033
ファイルのオープン，クローズ	open，close，nanosleep	9.35966
配列参照によるメモリアクセス	nanosleep	3.97767
usleep(20000) によるスリープ	nanosleep	3.95809

アルゴリズム 2 検証プログラムの概要

```
Input: char buf[8]: 標準入力
 1: procedure PowerProg
 2:     loopcnt = buf[2] − buf[3] − buf[4] − buf[5]
 3:     if loopcnt < 1 then
 4:         loopcnt = 1
 5:     end if
 6:     if (buf[0]&0x80) ≠ 0 then
 7:         if (buf[0]&0x80) ≠ 0 then
 8:             DiskRead(loopcnt)
 9:         else if (buf[0]&0x40) ≠ 0 then
10:             DiskWrite(loopcnt)
11:         else
12:             DiskMmap(loopcnt)
13:         end if
14:     else if (buf[0]&0x40) ≠ 0 then
15:         if (buf[0]&0x80) ≠ 0 then
16:             MemoryBrk(loopcnt)
17:         else
18:             MemoryMmap(loopcnt)
19:         end if
20:     else
21:         if (buf[0]&0x80) ≠ 0 then
22:             ClientRead(loopcnt)
23:         else
24:             ClientWrite(loopcnt)
25:         end if
26:     end if
27: end procedure
```

た．検証プログラムを永続的にループするようにし，ループの末尾に 20 ミリ秒ごとの usleep() を行うように変更した．変更した検証プログラムに対して，初期のシード，および最後に追加されたシードを入力として，それぞれ 5.1 節と同様の方法で消費電力を記録し，式 (1) にしたがって計算を行った．結果として，初期のシードでは 6.01366mW，最後に追加されたシードでは 104.41555mW となり，初期のシードと比較して，約 17.4 倍の消費電力と推定されるシードとなった．

6 議論

6.1 提案手法と実装の差異

本稿における提案手法は，ファジングによる解析対象プログラムの実行中に，消費電力測定プログラムを用いて消費電力を測定するものであった．それに対して，PoCoFuzz のプロトタイプでは，最も多くの消費電力経路を経由する入力を発見したのち，消費電力測定プログラムを用いて消費電力を測定した．また，消費電力測定プログラムを用いた消費電力モデルの作成においても，30 分の消費電力測定結果を用いて作成している．

これは，消費電力測定プログラムとして使用した PowerTOP による，1 実行時間あたりの消費電力の測定が困難であるからだ．PowerTOP では，標準の設定で，1 回の表示までに 20 秒の時間を要しており，20 秒間の消費電力を秒あたりの消費電力に整形して表示している．1 実行時間あたりの消費電力測定結果を利用することも検討したが，PowerTOP で表示される消費電力の幅自体が大きいことが分かり，PowerTOP 内で行われているパラメータの学習が不安定であると判断し，標準の 20 秒を利用することとした．

以上のことから，今回作成した PoCoFuzz のプロトタイプは，提案手法との差異が存在する．1 実行時間あたりの消費電力の測定が可能な消費電力測定プログラムが存在し，またそれを利用した場合，提案手法と同様の実装を行うことができ，より高精度な消費電力の測定を行うことが可能である．

6.2 実行速度と消費電力要因の追跡のトレードオフ

今回，動的解析として ptrace を使用した．ptrace の代替として，動的バイナリ計測フレームワークである Intel Pin[5] が考えられるが，Intel Pin を使用して PoCoFuzz を実装した場合，ptrace と比較して数十倍の実行速度の低下が発生した．同一の解析対象プログラムに対して，

表 2: 各ファザーの実行速度

ファザー	実行速度 [exec/s]
通常の AFL	9170.82
ptrace を使用した PoCoFuzz	1890.91
Intel Pin を使用した PoCoFuzz	25.08

動的解析を使用しない場合の AFL，ptrace を使用した場合の PoCoFuzz，Intel Pin を使用した場合の PoCoFuzz の実行速度をまとめた表を，表 2 に示す．

ptrace を使用した場合の PoCoFuzz が，Intel Pin を使用した場合の PoCoFuzz より実行速度の値が大きい．このことから、ptrace を使用した場合の PoCoFuzz の方が実行速度が優れている．以上より，今回は実行速度を優先し，ptrace を使用した PoCoFuzz を実装した．

6.3 静的解析と動的解析の組み合わせ

PoCoFuzz では，動的解析のみを使用して消費電力要因の追跡を行っているが，2.1 節で説明した静的解析と動的解析を組み合わせる VUzzer のように，静的解析を組み合わせることで，より精度の高い消費電力要因の追跡が可能になる．

静的解析と動的解析を組み合わせた例として，以下のような方法を用いることで，より効率的に消費電力要因の追跡が可能となる．

1. 静的解析により，消費電力要因を特定する．
2. 特定した消費電力要因数を，基本ブロックのような特定の単位ごとにまとめる．
3. 動的解析により，遷移した基本ブロックごとに消費電力要因数を加算する．
4. 求めた消費電力要因数をファザーの情報収集プロセスと共有する．

6.4 消費電力要因数

消費電力要因は，PoCoFuzz において消費電力モデルに大きく関わる重要な要素である．消費電力要因が多ければ多いほど，大量の消費電力を発生させる入力を生成するのに役立つ．今回は，ディスクアクセス，メモリ確保，通信を消費電力要因として消費電力モデルを作成したが，命令実行数，CPU 使用時間といった消費電力要因を追加することで、より精度の高い消費電力要因の追跡が可能になる．

ただし，消費電力要因数を増やすのであれば、6.2 節で述べたような実行速度とのトレードオフについて考える必要がある．表 2 の，通常の AFL と ptrace を使用した場合の PoCoFuzz を見比べると，ptrace を用いたシステムコールのみを追跡する簡素な動的解析においても，通常の AFL の約 4.8 倍遅い実行速度となっている．

7 おわりに

本稿では、外部からの入力に依存するプログラムに対して，電子機器ごとに調整された消費電力モデルを使用することで、消費電力の大きい入力を自動的に発見する PoCoFuzz を提案した．本提案手法により，従来不可能であった，消費電力の大きい入力を自動的に発見することが可能となった．これにより，消費電力の大きくなる影響要因を発見し，省電力なアプリケーションの開発の一助となることが可能となる．

提案手法について，実際に影響要因の動作を行うプログラム群を作成し，PowerTOP を用いて消費電力を測定した．測定した消費電力から，どの影響要因がどの程度の影響度を与えるか調査し，消費電力モデルの作成を行った．また，作成した消費電力モデルに従って，PoCoFuzz によるファジングを実行し，検証プログラムで最も大きい消費電力経路を経由する入力を発見することに成功した．最も大きい消費電力経路を経由する入力は，初期のシードと比較して 17.4 倍消費電力が大きいと推定される入力であった．

今回は，消費電力モデルの作成までにかかる時間や，ファジングの実行速度など，多くの課題が残った．今後は、これらの課題に対処するとともに、より効率的に消費電力の大きい入力を自動的に発見できるよう，改良を行っていく．

参考文献

[1] libFuzzer – a library for coverage-guided fuzz testing. — LLVM 8 documentation. `https://llvm.org/docs/LibFuzzer.html` (参照日: 2018/12/17).

[2] Hayri Acar, Gülfem I Alptekin, Jean-Patrick Gelas, and Parisa Ghodous. The Impact of Source Code in Software on Power Consumption. *International Journal of Electronic Business Management*, 14:42–52, 2016.

[3] intel. PowerTOP — 01.org. `https://01.org/powertop/` (参照日: 2018/12/17).

[4] lcamtuf. american fuzzy lop. `http://lcamtuf.coredump.cx/afl/` (参照日: 2018/12/17).

[5] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic

instrumentation. *SIGPLAN Not.*, 40(6):190–200, June 2005.

[6] B.P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. In *Communications of the ACM 33*, 12(1990).

[7] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. *CoRR*, abs/1708.08437, 2017.

[8] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware Evolutionary Fuzzing. In *NDSS*, February 2017.

[9] Saleh Soltan, Prateek Mittal, and H. Vincent Poor. Blackiot: Iot botnet of high wattage devices can disrupt the power grid. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 15–32, Baltimore, MD, 2018. USENIX Association.

[10] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, 2016.

[11] 神山剛, 稲村浩, and 太田賢. 電力モデルに基づくアプリ消費電力可視化ツールの評価. In マルチメディア、分散協調とモバイルシンポジウム *2013* 論文集, volume 2013, pages 286–292, jul 2013.