情報処理学会論文誌 Vol.62 No.9 1511–1522 (Sep. 2021)

推薦論文

IoTPoCoFuzz：消費電力を考慮したモデルベースのリモートファジングの実現とその半自動化

水野 慎太郎[1,a)] 西垣 正勝[1,b)] 大木 哲史[1,c)]

受付日 2020年11月30日, 採録日 2021年6月7日

概要：多種多様な機器・サービス・プロトコルが混在する環境下では，脆弱性だけではなく，望ましくないイベントの種類も多角化していく．望ましくないイベントの一種である消費電力の増大は，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．特に，これらの要因のうち，外部から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，入力と消費電力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．本論文では，機器の消費電力の大きい入力を，リモートから効率的に発見する手法を提案する．本手法は，消費電力を考慮したモデルベースのリモートファジングと，ネットワークトラフィックのキャプチャ結果からファジングで使用する定義ファイルの自動生成技術を用いることで，消費電力の大きい入力を外部から半自動で効率的に発見することを可能とする．本論文では，提案手法の動作原理について述べるとともに，実験を通して提案手法の信頼性と有効性を示す．

キーワード：消費電力，ファジング，自動化，プロトコル

IoTPoCoFuzz: Realization and Semi-automation of Power Consumption-aware Model-based Remote Fuzzing

Shintaro Mizuno[1,a)] Masakatsu Nishigaki[1,b)] Tetsushi Ohki[1,c)]

Received: November 30, 2020, Accepted: June 7, 2021

Abstract: Under the environment where a wide variety of devices, services and protocols, not only vulnerabilities but also unintentional events will be diversified. Power consumption, which is a kind of unintentional event, has diverse factors of increase. Therefore, it is difficult to identify the cause of the increase, and has a variety of ranges of impact. In particular, among these factors, the input-dependent processing of external depends on the input. In that case, estimated power consumption is very difficult that be a complicated relationship between input and power consumption. In this paper, we propose a method to find an input that causes significant power consumption of devices efficiently through an external network. The method be combined power consumption aware model-based remote fuzzing, namely, IoTPoCoFuzz. The method automatically makes definition file for fuzzing from network traffic. Then it efficiently finds an input that causes significant power consumption efficiently through an external inputs. Our paper describes the operating principle of the proposal method and shows effectiveness and reliability through an experiments.

Keywords: power consumption, fuzzing, automation, protocol

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．脆弱性の種類は多岐にわたり，事実，プログラムの脆弱性は近年においても多数報告され

[1] 静岡大学
Shizuoka University, Hamamatsu, Shizuoka 432–8011, Japan
[a)] mizuno@sec.inf.shizuoka.ac.jp
[b)] nisigaki@inf.shizuoka.ac.jp
[c)] ohki@inf.shizuoka.ac.jp

本論文の内容は 2020 年 10 月のコンピュータセキュリティシンポジウム 2020（CSS2020）にて報告され，同プログラム委員長により情報処理学会論文誌ジャーナルへの掲載が推薦された論文である．

© 2021 Information Processing Society of Japan

ている [1]．ファジングは対象ソフトウェアへの入力値を変化させながら脆弱性を発見するテスト手法であり，従来は開発者の経験によって行っていた手動による脆弱性テストを自動化することを目指している．一方，IoT 機器をはじめとした，多種多様な機器・サービス・プロトコルが混在する環境下では，脆弱性だけではなく，実行速度の低下やメモリ使用量の増加といった望ましくないイベントの種類も，環境・サービス・プロトコルごとに多角化していくと考えられる．このような望ましくないイベントの 1 つとして，機器の電力消費があげられる．電力の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，バッテリ駆動する電子機器の場合，バッテリ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．特に 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，電源供給設備に対する意図的な停電が引き起こされる可能性が指摘されており [2]，事実，2016 年には，ウクライナにおいて制御装置の脆弱性を悪用するツールを含んだマルウェアが停電を引き起こしている [3]．

このような背景をふまえ，本論文は，IoT 機器を対象とした，消費電力を増大させる入力を効率的に発見するファジング手法である IoTPoCoFuzz を提案する．消費電力の増大要因は，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多岐にわたり，その発見は非常に困難である．特に，これらの要因が外部からの入力に依存する場合，その結果が入力内容に依存することにより，入力と電力消費の要因との関係はいっそう複雑となる．これまでも Acar ら [4] や神山ら [5] によって，CPU やディスクといった機器の構成要素の動作と消費電力の関係をモデル化する消費電力の推定方法が提案されてきたものの，入力への依存性は明らかにされいない．本研究が対象とする IoT 機器の多くは，その特性からリモートからネットワークを介したデータ入出力を行うことを前提としている．このため，その消費電力は入力情報に強く依存すると考えられる．リモートから入力を行うファジングとしては Boofuzz [6] や Peach [7] などが提案されているが，これらは脆弱性の発見を目的としており，消費電力をはじめとする望ましくないイベントを発見することができない．

さらに，ネットワークサービスでの通信を含む機器へのファジングを行う際には，通信する機器間で用いられるプロトコルのフォーマットを定義した情報（以下，プロトコルフォーマットと呼ぶ）などを含めた，ファジングの実行に必要な情報を定義する必要があるが，特にプロトコルフォーマットの作成には，対象とするシステムに対する専門的な知識が必要であり，これがファジング利用への障壁となっていた．プロトコルフォーマットを自動生成するファジング手法は，これまでいくつか提案されている [8], [9], [10] が，Boofuzz や Peach と同様，脆弱性を発見することしか考慮されていない．

そこで本提案手法では，ネットワークトラフィックから推定したプロトコルフォーマットを用いてファジングに必要な定義ファイルを自動的に作成する．作成した定義ファイルに基づき IoT 機器へ入力する情報を適切に変異させることで，リモートから IoT 機器のファジングにより消費電力の増大する入力を効率的に発見することを可能とする．

まず，大きい電力消費に関連する要因を含むプログラムの一連の動作（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，本手法が想定する消費電力に関連する要因（以下，消費電力要因と呼ぶ）と消費電力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，ネットワークトラフィックを監視し，トラフィック中のパケットから，プロトコルフォーマットとプロトコルフォーマット中に含まれるトークン（特定のルールに従って記載された文字列）ごとの変異手法を適切に選択することで，ファジングに使用する定義ファイルを自動生成する．最後に，消費電力モデルとファジングを組み合わせることで，リモートから消費電力経路を経由する入力を効率的に発見する．

本論文が提案する IoTPoCoFuzz は，リモートで動作する機器に対して，半自動的かつ効率的に消費電力の大きい入力を効率的に発見することが可能である．提案手法の有用性評価では，提案手法の対象とする機器に該当する RaspberryPi を対象として，消費電力モデルにモデル化する消費電力要因による，最終的な消費電力の違いを分析する．また，RaspberryPi 上で動作する複数のソフトウェアに対して提案手法を実行することで，効率的に消費電力の大きい入力を発見できること，発見した入力が同一環境の他機器に対しても有効であることを示す．

本研究の貢献は，次のようにまとめられる．

- 消費電力の大きい入力をリモートから，半自動かつ効率的に発見する手法を実現した．
- 複数機器，かつ複数のオープンソースソフトウェアを用いて提案手法の有効性を検証した．

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探索的なテスト手法の一種である．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り，遺伝的アルゴリズムや進化的アルゴリズムなどを用いるもの，他テスト手法と組み合わせる

© 2021 Information Processing Society of Japan

ものなど，様々な手法が存在する [11], [12], [13]．しかし，近年では，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 [14] など，本来意図しない動作を発見する手法へとその応用範囲が広がりつつある．

ネットワークを通じて，ファザーが動作する機器とは異なる機器に対してファジングを行うリモートファジング手法も多く存在する．リモートファジングの多くは，ファジングで生成する入力のフォーマットとなるモデルや通信方式などを定義する必要がある [6], [7]．たとえば，Boofuzz はプログラム中に，Peach は XML 形式で記述する Pit ファイルに，それぞれ定義する．このようにして定義したモデルに基づいて入力を生成するファジングの手法は，モデルベースファジングと呼ばれる．リモートファジングでは，ネットワークを通じてファジングを行う性質から，モデルベースファジングでもあることが多い．ネットワークのほかにも，SPIKEfile [15] のようなファイルを対象としたもの，syzkaller [16] のようなカーネルシステムコールを対象としたものなどが存在する．

2.2 消費電力の推定

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．Acar ら [4] 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，それらをもとに，消費電力を見積もるツールを紹介した．神山ら [5] 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，アプリケーション内のページ遷移や操作内容といったアプリケーションごとに収集したログと，電力モデルの要素ごとの計算に必要なログを組み合わせることで，アプリケーションの消費電力を推定している．

Acar 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ている．これに対して，PowerTOP [17] は，要素ごとの電力モデルに加えて，ftrace と呼ばれるトレース機構を使用した，プロセスごとのコンテキストスイッチや割込み要求といったイベントの追跡によって，プロセスごとの消費電力を推定することを可能としている．

これら 3 つの手法は，入力に依存しない平均的な電力消費量の推定を主眼においており，入力に依存したプログラムにおいて，入力値によって最大消費電力が変化する点が考慮されていない．それに対して，本論文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，消費電力が増大する入力値の発見を可能とする．

2.3 モデルの自動生成

2.3.1 ネットワークトラフィックを用いたモデルの自動生成

モデルベースファジングを実施する場合，公開されている定義済みのモデルを使用することや，ユーザが自身でモデルを作成することが一般的であった．一方，近年では，モデルを自動で定義する手法が注目されている．Hsu ら [8] は，キャプチャしたネットワークトラフィックから，プロトコルの状態遷移モデルと，各状態におけるメッセージを推論した．推論結果をもとに自動生成したモデルを使用することで，モデルベースファジングの自動化を可能とした．状態遷移モデルとメッセージの変異に際しては，状態遷移モデルの現状態を変更する，メッセージの文字列を膨大な文字列にする，メッセージの数値を小数にするなど，プログラムに問題が発生すると思われる変異手法を無作為に選択することで，システムをクラッシュさせた．AutoFuzz [9] は，ネットワークトラフィックからの状態遷移モデルやメッセージの推論に加え，メッセージの変異箇所を推論したメッセージテンプレートを用意しておくことで，メッセージの変異箇所を限定している．

これらの手法は，いずれも脆弱性の発見を目的とし，システムにとっての異常な通信データを入力とすることを前提としている．一方，望ましくないイベントは，異常な通信だけでなく，正常な通信時においても発生する可能性がある．そこで，従来手法に対し，型の推論や適切な変異手法の選択，通信手法の工夫によって，望ましくないイベントを考慮可能なモデル自動生成手法を提案し，これを提案手法に適用することで，効率的に消費電力の大きい入力を発見するファジング手法を実現する．

2.3.2 PULSAR

PULSAR [10] は，AutoFuzz と同様に，ネットワークトラフィックから状態遷移モデルや各状態におけるメッセージのテンプレートを含んだモデルを推論に基づいて自動生成することで，モデルベースファジングの自動化を可能としている．PULSAR のモデル推論に際しては，PRISMA [18] に基づいている．その手順の概要を以下に示す．

(1) **メッセージをセッション群に分割する**：モデル推論の対象とするネットワークトラフィックのキャプチャデータから，一定時間内に送受信されたパケットをメッセージとしてまとめる．このとき，メッセージはクライアント/サーバのどちらから送信されたものであるかを示す注釈をつけておく．また，一定時間内に送受信しているメッセージ群をセッションと見なし，セッションごとに識別子を付与する．

(2) **メッセージのクラスタリングを行う**：テキストベースのプロトコルの場合はトークン，バイナリベースのプロトコルの場合は n-gram をそれぞれデータと見なし，クラスタリングを行う．PRISMA によるクラスタリ

© 2021 Information Processing Society of Japan

ングでは，各メッセージがどのクラスタに属するか判別する．

(3) **ネットワークプロトコルの状態遷移を模倣する**：手順 (1) のセッション識別子と手順 (2) のクラスタリング結果をもとに，状態遷移モデルを生成する．状態遷移モデルは，各状態の遷移確率を計算することで，マルコフモデルとしてモデル化する．

(4) **メッセージを生成する**：手順 (1) のメッセージと手順 (3) の状態遷移モデルをもとに，状態遷移ごとのメッセージテンプレートを生成する．同じクラスタのメッセージ間において，データの異なる箇所があった場合，そのデータが可変であると判定する．プロトコルフォーマットには，どの状態遷移時のものであるか，どのデータが可変か，といった情報を付与する．

IoTPoCoFuzz では，PULSAR のモデル推論手法に基づいてネットワークトラフィックを用いたプロトコルフォーマットの推論を行う．

3. 提案手法

3.1 提案手法概要

IoTPoCoFuzz の概要を**図 1** に示す．IoTPoCoFuzz は，まず，(a) 消費電力に関連すると思われる要因に関して，要因と消費電力の関係を明らかにし，消費電力モデルとしてモデル化する．次に，(b) ファジング実行機器とファジング対象機器間でのネットワークトラフィックをキャプチャし，(c) 定義ファイルを自動的に生成する．最後に，(d) 自動生成した定義ファイルを用いてファジングを実行することで，消費電力の大きい入力を発見する．

3.2 消費電力モデルの作成

本節では，消費電力モデルの作成手法の詳細を述べる．**図 2** は，図 1 における手順 (a) 消費電力モデルの作成の詳細を示したものである．

3.2.1 消費電力要因値の推定

消費電力は，呼び出されるシステムコールの種類，現実の実行時間，命令実行数など，様々な要因が関係する．これらを消費電力要因の集合を $f = \{f_i\}_{i=1\sim n}$ と定義する．各消費電力要因 f_i が与える消費電力への影響を，消費電力要因値 p_{f_i} として計測することで，各要因が消費電力に与える影響を推定する．

消費電力要因値を測定する際は，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，要因 f_i ごとに単一の処理による負荷を定期的に与えるプログラムを作成し，実行・測定を行う．ここで，システムコールの消費電力要因値は，read や write といった各システムコールの呼び出し時に消費される電力に基づき算出される．消費電力要因値は機器が消費する電力に基づき算出することで高精度に計測を行うことができる．一方，機器へのアクセスが制限されている状況においては，ファジング対象機器自体やプログラムごとの消費電力の測定ができない場合がある．このような状況においては，電力の代わりに，現実の実行時間や命令実行数など，実行時間・実行数といった計測値に基づき算出される消費電力要因を用いる必要がある．そのため，便宜上，消費電力要因タイプ $type_{f_i} \in \{0,1\}$ を定義し，消費電力要因値を消費される電力に基づき算出するものを $type_{f_i} = 0$，計測値に基づき算出するものを $type_{f_i} = 1$ として定義する．

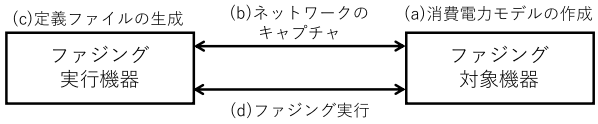

図 1 IoTPoCoFuzz による消費電力の大きい入力の発見手法
Fig. 1 Method of detecting inputs with high power consumption by IoTPoCofuzz.

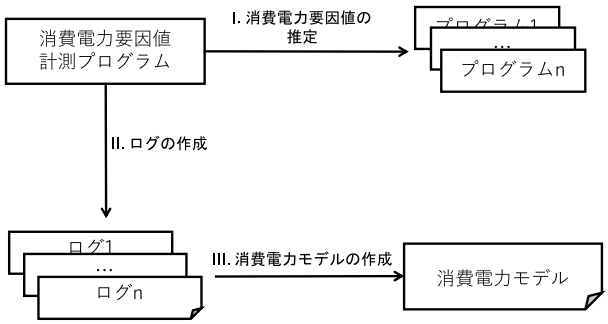

図 2 消費電力モデルの作成手順
Fig. 2 Procedure for creating a power consumption model.

3.2.2 消費電力要因値ログの作成

要因 f_i に対応した消費電力要因値計測プログラムを作成し，実行する．実行したプログラムにより計測された消費電力要因値から消費電力要因値ログを作成する．

3.2.3 消費電力モデルの作成

時刻 t における消費電力要因 f_i がとる消費電力要因値を $p^t_{f_i}$ としたとき，f_i の単位時間あたりの消費電力要因値の期待値 $\bar{p}_{f_i}$ は次式で求められる．

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

f_i は消費電力要因を表しており，f_i ごとに消費電力要因値の期待値 $\bar{p}_{f_i}$ を求める必要がある．また，$\bar{p}_{f_i}$ は電力と測定値のタイプが存在するため，消費電力要因値のタイプ $type_{f_i}$ を用意する必要がある．また，本提案では，usleep（100）の処理を繰り返し行うプログラム実行時の電力を定常状態と仮定する．なお，定常状態を測定する方法は多くの方法が考えられるが，本論文においては，消費電力要因の動作を行うプログラムに追加しやすく，消費電力要因の動作による消費電力差が顕著に現れるという観点から usleep を選択し，引数については消費電力差が表れる数値を実験的に選択した．ここで，定常状態における単位時

© 2021 Information Processing Society of Japan

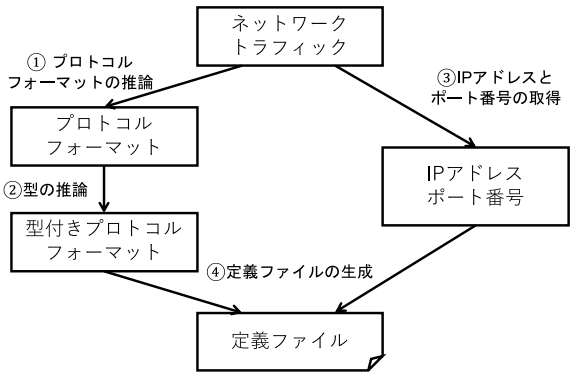

図 3 定義ファイルの自動生成手順
Fig. 3 Procedure for automatic generation of definition file.

間あたりの消費電力の期待値を $\bar{p}_{usleep}$ とし，消費電力モデルに追加することで定常状態を基準とした入力の探索を可能とする．消費電力モデル M は，これらの値を用いて，

$$M = \{M_{f_1}, M_{f_2}, \cdots, M_{f_i}, \cdots, M_{f_n}, \bar{p}_{usleep}\} \quad (2)$$

$$M_{f_i} = \{f_i, \bar{p}_{f_i}, type_{f_i}\} \quad (3)$$

と表される．

なお，システムコールなどは消費電力要因が read システムコールや write システムコールなど，複数の要素から構成される．このような場合は，消費電力要因を複数の子要素に分割して，それぞれの子要素に対して本モデルを再帰的に作成する．

3.3 定義ファイルの自動生成

図 1 のうち，手順 (b) ネットワークトラフィックデータから手順 (c) 定義ファイルの生成までの詳細を**図 3** に示す．まず，ネットワークトラフィックをキャプチャしたデータからプロトコルフォーマットを推論（①）し，プロトコルフォーマットの各トークンの型を推論（②）する．あわせて，あらかじめ定義されたトークンの値や型ごとの変異手法を設定する．推論した型付きプロトコルフォーマットの生成後，キャプチャデータからファジング対象機器の IP アドレスとポート番号を取得（③）し，型付きプロトコルフォーマットと組み合わせることで，定義ファイルを自動的に生成（④）する．

3.3.1 プロトコルフォーマットの推論

プロトコルフォーマットの推論は，2.3.2 項に記載した PULSAR のモデル推論手法に基づいており，手順はアルゴリズム 1 に示すとおりである．アルゴリズム 1 の各関数のうち，2–5 行目は，2.3.2 項で述べた PULSAR によるモデル推論手順に対応しており，ネットワークトラフィックデータ NT から，状態遷移モデル $state$ と，状態遷移ごとのメッセージテンプレート $template$ を生成する．6 行目の CreateProtocolFormat は，2.3.2 項で述べた PULSAR によるモデル推論手順 (4) の後に行われる，本提案手法で新たに追加された処理である．望ましくないイベントである消費電力の増大は，正常な通信と異常な通信の両者で発生する可能性があるため，正常な通信と異常な通信の両者を検証可能とする必要がある．そのため，CreateProtocolFormat では，状態遷移モデルとメッセージテンプレートからプロトコルフォーマット PF を生成する．

アルゴリズム 1 ネットワークトラフィックデータ NT から，プロトコルフォーマット PF を生成する

```
1: procedure ModelInference(NT)
2:     sessionData = SessionExtract(NT)
3:     clusterData = Clustering(sessionData)
4:     state = CreateStateMachine
                        (sessionData, clusterData)
5:     template = CreateTemplate(sessionData, state)
6:     PF = CreateProtocolFormat(state, template)
7:     return PF
8: end procedure
```

表 1 トークンに対する型の推論と変異手法の選択例（default は型ごとに用意されている標準の変異手法を表す）
Table 1 Examples of mutation method and type of inference for each token: “default” represents a standard mutation method preapred for each type.

トークン	推論される型	選択される変異手法
GET	文字列	default
\index.html	文字列	default，パス
192.168.0.1	文字列	default，IP アドレス
1000	数値	default
0x0bac9001	バイナリ	default

3.3.2 型の推論

型の推論では，数値・文字列・バイナリといったトークンの型判別を行う．その後，トークンの型やその値に基づき，適切な変異手法を選択する．

トークンの値に対する，型の推論と変異手法の選択例を**表 1** に示す．まず，トークンの値をもとに，数字列のみであれば数値，文字が含まれていれば文字列，数値や文字以外のバイナリ列が含まれていればバイナリと推論する．次に，型ごとに用意されている標準の変異手法である default に加えて，トークンがディレクトリのパスや IP アドレスなどに類似している値であれば，追加で変異手法を選択する．表 1 では，\index.html がディレクトリのパス，192.168.0.1 が IP アドレスに類似していると判定され，それぞれ変異手法が追加されている．

アルゴリズム 2 に型推論アルゴリズムを示す．プロトコルフォーマットからリクエストメッセージテンプレートを 1 つずつ取り出し，型の推論を行う（8–13 行目）．メッセージテンプレートからトークンを取り出し（9 行目），トークンごとに TokenTypeInference による型の推論（10 行

© 2021 Information Processing Society of Japan

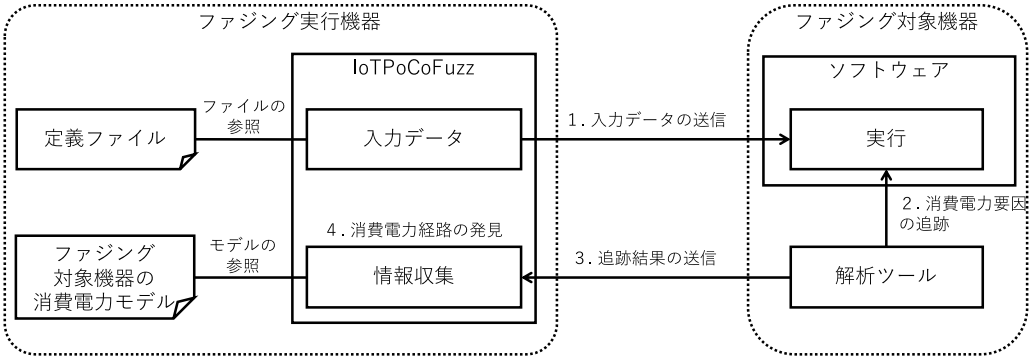

図 4 IoTPoCoFuzz による消費電力要因の追跡
Fig. 4 Tracking power consumption factors by IoTPoCoFuzz.

アルゴリズム 2 プロトコルフォーマット PF から，型付きプロトコルフォーマット TPF を生成する

```
1: procedure TypeInference(PF)
2:     TPF = ∅
3:     aid = 0
4:     allPFSize = Size(PF)
5:     while aid < allPFsize do
6:         tid = 0
7:         templateSize = Size(PF[fid])
8:         while tid < templateSize do
9:             token = PF[aid][tid]
10:            token.type = TokenTypeInference(token)
11:            token.mutate = SelectMutate(token)
12:            Append(TPF, aid, tid, token)
13:            tid = tid + 1
14:        end while
15:        aid = aid + 1
16:    end while
17:    return TPF
18: end procedure
```

目）と SelectMutate による適切な変異手法の選択（11 行目）を行い，型付きプロトコルフォーマットへ追加（12 行目）する．

3.4 消費電力要因の追跡

図 1 のうち，手順 (d) ファジングの実行の詳細について説明する．IoTPoCoFuzz によるファジングでは，消費電力モデルを参照しながらファジング対象機器に対する消費電力要因の追跡を行うことで，消費電力の大きい入力を発見する．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 4 に示す．

図 4 の主な流れは，以下のとおりである．

1. 入力データの送信：定義ファイルに基づき入力データを生成し，ファジング対象機器のソフトウェアに送信する．
2. 消費電力要因の追跡：入力データの送信後から，ソフトウェアが入力データに対する一連の処理を実行し終わるまでの間，解析ツールを用いて消費電力要因を追跡する[*1]．外部からの入力を受け付けるソフトウェアの場合，入力データの送信前と一連の処理の後に，新しい入力データを受け付ける処理があるため，その処理を監視することで，追跡の開始と終了を判定する．
3. 追跡結果の送信：消費電力要因の追跡の結果得られたデータを，IoTPoCoFuzz に送信する．
4. 消費電力経路の発見：ファジング対象機器の動的解析から送信された追跡結果をもとに，消費電力経路を発見する．発見に際しては，ファジング対象機器の消費電力モデルを参照する．消費電力経路が発見された場合，発見時の入力データを以降の入力データ生成に役立てる．

各構成要素を使用した IoTPoCoFuzz の主要なアルゴリズムを，アルゴリズム 3 に示す．IoTPoCoFuzz では，ファジング対象機器との接続（4 行目）を行った状態で，ファジングを開始する（5–14 行目）．定義ファイルと入力のもととなるシード集合からシードを 1 つ選択（6 行目）し，シードを変異させる（7 行目）．変異したシードを対象機器への入力とした状態で，ファジング対象機器への入力を行う（8 行目）．ファジング対象機器への入力後，対象機器の消費電力要因の集合 $PFS = \{PFS_j\}_{j=1\sim x}$，および $type_{PFS_j}$ が 1（計測値）である消費電力要因 PFS_j 追跡時の計測値 $FM = \{FM_{PFS_j}\}_{type_{PFS_j}=1}$ を解析ツールによって追跡することで，追跡結果 $power\_data = \{PFS, FM\}$ を得る（8 行目）．ここで，本論文において，計測値 FM は消費電力と比例の関係を示すように選択した．追跡結果から消費電力経路を判定する値（以下，消費電力経路値 $PCRS$ と呼ぶ）を次式 (4) により算出する．

$$PCRS = \sum_{PFS_j \in PFS} \sum_{f_i \in f} PR_{[PFS_j = f_i]} \tag{4}$$

[*1] ファジング中には消費電力の測定を行わないため，解析ツールは消費電力に影響を及ぼさない．

© 2021 Information Processing Society of Japan

アルゴリズム 3 対象機器 D に対して，定義ファイル DF およびシード集合 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 IoTPoCoFuzz(D, DF, S, n)
2:     power_units = ∅
3:     cnt = 0
4:     Connect(D, DF)
5:     while cnt < n do
6:         in =SelectSeed(DF, S)
7:         in′ =MutateInput(DF, in)
8:         power_data =SendInput(D, in′)
9:         if CheckPCRS(power_data) then
10:            S∪ = in′
11:            power_units∪ = in′
12:        end if
13:        cnt = cnt + 1
14:    end while
15:    Disconnect(D)
16:    return power_units
17: end procedure
```

ここで，式 (4) の $PR_{[PFS_j=f_i]}$ は，式 (5) によって求められる．

$$PR_{[PFS_j=f_i]} = \begin{cases} \bar{p}_{[PFS_j=f_i]} - \bar{p}_{usleep}, \\ \qquad type_{[PFS_j=f_i]} = 0 \\ \dfrac{FM_{[PFS_j=f_i]}}{\bar{p}_{[PFS_j=f_i]}} \times \bar{p}_{usleep}, \\ \qquad type_{[PFS_j=f_i]} = 1 \end{cases} \tag{5}$$

$PCRS$ の値が大きいほど，消費電力の大きい消費電力経路であると判定する．ここで，$PCRS$ は，定常状態からの消費電力の増減を示す指標とするため，式 (5) において，定常状態との差分もしくは比率を求める．消費電力要因のタイプ $type_{PFS_j}$ が 0（電力）である場合，消費電力要因値は消費電力をもとに算出されるため，定常状態との差分を算出する．対して，$type_{PFS_j}$ が 1（計測値）である場合，計測値をもとに，$PCRS$ の目的にあった定常状態からの消費電力の増減を示す指標とするため，定常状態との比率を算出する．

追跡結果を確認し，これまでで最も $PCRS$ が大きい入力を，消費電力経路値で実行していると思われる入力だと判断してシード集合に追加する（9–12 行目）．消費電力経路を実行していると思われる入力をシード集合に追加して，今後の入力データ生成に利用することで，より消費電力の大きい入力を効率的に発見することが可能となる．最後に，対象機器との接続を終了（15 行目）し，消費電力経路を経由する入力の集合を返す（16 行目）．

4. 実装

提案手法のプロトタイプの実装にあたり，消費電力測定値計測プログラムは PowerTOP と perf，モデルの推論は PULSAR [*2] および自作の Python プログラム，型の推論は自作の Python プログラム，ファザーは Peach [*3]，解析ツールは strace と perf を使用した．

消費電力モデルの作成に関する実装では，3.2 節に従い，PowerTOP を用いて消費電力を，perf を用いて計測値を，それぞれ消費電力測定値として計測する．計測したデータは消費電力測定値ログとした後，自作のシェルスクリプトを用いて消費電力測定値ログを消費電力モデルに変換する．

定義ファイルの自動生成に関する実装では，アルゴリズム 1 に従い，状態遷移モデルおよびメッセージテンプレートを PULSAR によって生成し，状態遷移の開始時に使用されるメッセージテンプレートを自作 Python プログラムによりモデル化する．型の推論では，アルゴリズム 2 に従い，型の推論や変異手法の選択を行うことで，型付きプロトコルフォーマットを自動生成する．最後に，型付きプロトコルフォーマットに対して，手動で IP アドレスとポート番号を組み合わせることで定義ファイルを作成する [*4]．なお，型の推論では，変異手法として Peach で定義されている変異手法を中心に扱うが，IoTPoCoFuzz 独自の変異手法に関しても，変異手法選択時の対象とする．

ファザーに関する実装では，解析ツールによって得られた情報をもとに，消費電力経路を経由する入力をシードへ追加する．strace を用いた解析では，各ソフトウェアで動作するシステムコール群を追跡する．たとえば，lighttpd の場合，データの送受信であれば read システムコール，もしくは sendfile64 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，システムコールの動作を判別する．perf を用いた解析では，perf_event_open システムコールを通じて perf の測定項目を取得する．また，今回の実験に使用する HTTP プロトコル，MQTT プロトコルへ対応するように軽微な修正を施した．なお，IoTPoCoFuzz では，消費電力経路を発見する性質から，異常なリクエストと同様，正常なリクエストについても検証を行う必要がある．しかし，通常のファザーは異常なリクエストが生成するリクエストの大半を占める．このため，本提案手法では，特定のディレクトリ内のファイルリストから任意のファイル名を 1 つ返す，という変異を用意した．これにより，ファイルを必要とするプログラムに対して，正常なファイルを送信することが可能となる．

[*2] https://github.com/hgascon/pulsar
[*3] https://github.com/MozillaSecurity/peach
[*4] 実験・検証で複数台のファジング対象機器を扱うため手動で設定したが，キャプチャデータからファジング対象機器の IP アドレスとポート番号を取得するプログラムを作成することで，自動的に設定することが可能となる．

© 2021 Information Processing Society of Japan

5. 実験・検証

本実験は,以下の3点の実験・検証を通して,IoTPoCoFuzzの有効性と信頼性を示すことを目的とする.

- 利用する消費電力要因を変更した場合の消費電力の違いを確認することで,IoTPoCoFuzzに適した消費電力要因を分析する.
- IoTPoCoFuzzにより生成された入力を用いて消費電力の増加を検証することで,IoTPoCoFuzzの有効性を確認する.
- IoTPoCoFuzzにより生成された入力を,同一環境の他機器でも有効であるかを調査することで,IoTPoCoFuzzの信頼性を確認する.

ここでは,IoT機器で使用される代表的なプロトコルとして,トークンベースのヘッダを用いるプロトコルであるHTTPと,バイナリベースのヘッダを用いるプロトコルであるMQTTを対象とした実験を行う.HTTPについては,軽量で組み込み用途にも使われるHTTPサーバであるlighttpdを,MQTTについてはメッセージを中継するサーバの役割を持つmosquittoとメッセージを受信するmosquitto_subを,それぞれ検証対象ソフトウェアとして用いる.

5.1 実験環境

すべての実験・検証において,ファジング実行機器はインテル(R) Core i7プロセッサと16 GBのメモリを搭載したmacOS10.13.6を使用し,ファジング対象機器および検証対象機器をRaspberryPi3 Model B+とした.ファジング対象機器および検証対象機器は,実験・検証に不必要なプロセスは手動で停止する,ファジングを実施するたびに手動で再起動する,といった簡易的な方法を用いて,他の要因による実験・検証への影響を抑えている.消費電力の測定に関しては,消費電力測定機器としてExtech 380803[*5]を使用した.ファジング対象機器と検証対象機器で使用するソフトウェアに関して,lighttpd,mosquitto,mosquitto_subを使用した.なお,ソフトウェアはキャッシュを使用せず実験を行った.検証実験においては,ファジング対象機器とは別個体のRasberryPi3 Model B+を用意し,ファジングによって発見した入力の有効性を検証した.なお,ファジングの実行時には,ファジング対象機器によって異なる乱数シードを用いている.

5.2 実験準備

5.2.1 ネットワークトラフィック

検証対象ソフトウェアの定義ファイルを自動生成するため,ファジング実行機器とファジング対象機器間でHTTP通信,MQTT通信を行い,それぞれの通信をキャプチャした.キャプチャした通信は,頻繁に発生することが想定される通信を模擬することを目的とし,HTTP通信はHTTPリクエストメソッドをGETとPOSTとしたリクエストとそれに対するレスポンスを,MQTT通信はテキストやファイルといったデータを送信するリクエストとそれに対するレスポンスを,それぞれキャプチャした.通信をキャプチャする際,ファジング実行機器とファジング対象機器間でのHTTP通信,もしくはMQTT通信のみをキャプチャの対象とし,対象外の通信はキャプチャしない.結果として,HTTP通信では6,169パケット,MQTT通信では25,199パケットをそれぞれキャプチャした.

5.2.2 データセット

実験・検証で使用するソフトウェアでは,HTTP通信のGETとPOSTを用いたファイルを要求するリクエストや,MQTT通信のファイルデータを送信するリクエストといったファイル操作が要求されるリクエストが存在する.ファイル操作に使用するファイルを統一するため,HTMLファイルと画像を含んだ10,000ファイルのデータセットを用意した.HTMLファイルは,MediaWiki API[*6]を使用して無作為に5,000ファイルを収集した.画像は,Indoor Scene Recognition[*7]から無作為に5,000ファイルを収集した.

5.3 消費電力モデルを構成する消費電力要因の分析

利用する消費電力要因による消費電力の影響を分析するため,システムコール,現実の実行時間,CPU実行時間,命令実行数を消費電力要因として,ファジング対象機器上で動作するlighttpdに対して,それぞれの消費電力要因を使用した場合の消費電力を計測する.

消費電力要因のうち,システムコールは,ファジング実行機器上で動作する検証対象ソフトウェアのシステムコールを事前に調査し,各システムコールを呼び出すプログラムを作成した.システムコールの動作を行うプログラムは,たとえばreadシステムコールのディスクへの読み込みやネットワーク通信の受信など,複数の用途があるシステムコールについては,その用途に応じてそれぞれプログラムを作成した.次に,作成したプログラムを実行し,それぞれの消費電力を30分間記録する.なお,このとき,0 mWといった極端に小さな値は定常雑音と見なして計算に用いないこととした.また,現実の実行時間,CPU実行時間,命令実行数は,それぞれusleep(100)を繰り返し実行するプログラムを実行し,それぞれの測定値を30分間記録する.最後に,記録した消費電力または測定値に基づき,式(4)により消費電力要因値を計算し,消費電力モデルを作成する.

[*5] http://www.extech.com/products/380803

[*6] https://www.mediawiki.org/wiki/API:Main_page/ja

[*7] http://web.mit.edu/torralba/www/indoor.html

© 2021 Information Processing Society of Japan

表 2 消費電力要因ごとの消費電力
Table 2 Power consumption value for each power consumption factor.

消費電力要因	lighttpd の消費電力 [mW]
システムコール	1.658
現実の実行時間	4.469
CPU 実行時間	1.677
命令実行数	1.511

それぞれの消費電力要因を，3.2 節に従って消費電力モデルにモデル化し，IoTPoCoFuzz によるモデルの自動生成とファジングを実行し，消費電力要因ごとに消費電力を増大させる入力を発見する．定義ファイルの自動生成に際しては，5.2.1 項でキャプチャしたネットワークトラフィックを用いた．発見した入力を lighttpd に対して 10 回入力し，入力した際の平均消費電力値を算出した．算出した消費電力要因ごとの消費電力を**表 2** に示す．表 2 から，現実の実行時間が最も消費電力を増大させる入力を発見できていることが分かる．これは，現実の実行時間が，他の消費電力要因では計測できないデバイスの応答などといった要素を間接的に計測可能であることなどに起因することが考えられる．

5.4 ファジングを用いた消費電力の増加に関する検証

5.4.1 各検証対象ソフトウェアに対するファジング

5.3 節で作成した消費電力モデルをもとに，ファジング対象機器上で動作する検証対象ソフトウェアに対して，IoTPoCoFuzz によるファジングを実施した．定義ファイルの自動生成に際しては，5.2.1 項でキャプチャしたネットワークトラフィックを用いた．また，ファジングの実施時，5.3 節の結果をもとに，ファジング対象機器の消費電力モデルを作成した．なお，5.3 節の結果をふまえ，ここでは消費電力要因として現実の実行時間を扱う．

図 5，**図 6**，**図 7** に，ファジング対象機器内で動作する検証対象ソフトウェアに対して IoTPoCoFuzz によるデータ入力を 10,000 回実行した際の，入力回数と入力回数ごとの最大消費電力経路値の関係を示す．それぞれの図では，縦軸が式 (4) に従って算出した各入力回数における最大消費電力経路値，横軸が入力回数を表す．これらの結果から，IoTPoCoFuzz によりすべてのソフトウェアに対して，初期の入力よりも大きい消費電力経路値が発見されていることが分かる．なお，IoTPoCoFuzz による 10,000 回の実行にかかる時間は，lighttpd は 41 分，mosquitto は 2 時間 28 分，mosqiutto_sub は 2 時間 31 分であった．

5.4.2 ケーススタディ：lighttpd

lighttpd に対する実行結果のうち，初期に生成された入力と最後に生成された消費電力の大きい入力を，それぞれ**図 8**，**図 9** に示す．両方の入力を比較すると，GET リクエストによってリクエストしているファイルが異なってい

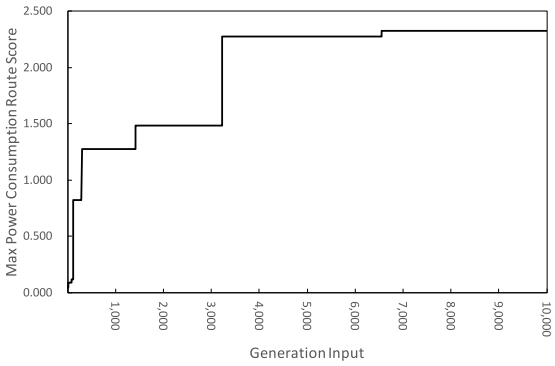

図 5 lighttpd に対する入力回数ごとの最高消費電力経路値
Fig. 5 Maximum power consumption route score per input count to lighttpd.

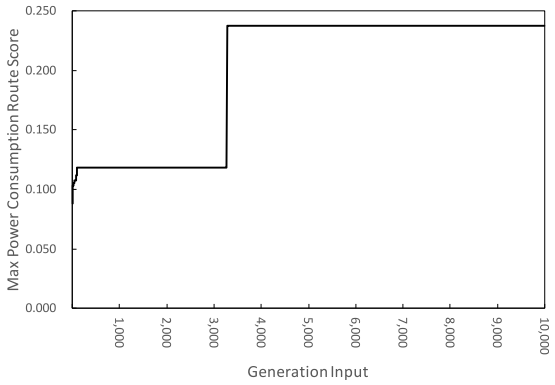

図 6 mosquitto に対する入力回数ごとの最高消費電力経路値
Fig. 6 Maximum power consumption route score per input count to mosquitto.

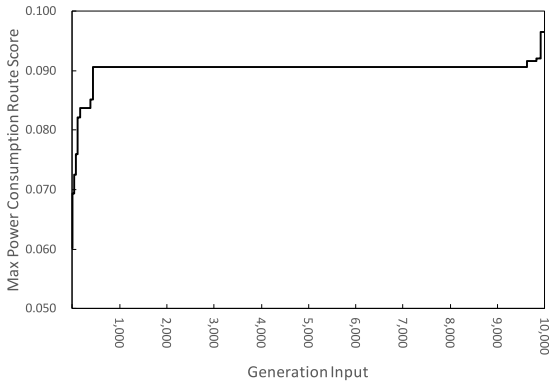

図 7 mosquitto_sub に対する入力回数ごとの最高消費電力経路値
Fig. 7 Maximum power consumption route score per input count to mosquitto_sub.

```
GET /dataset/字型車体V.html HTTP/1.1
Host: 192.168.0.8
User-Agent: curl/7.54.0
Accept: */*
```

図 8 IoTPoCoFuzz によって最初に生成された入力
Fig. 8 An initial input generated by IoTPoCoFuzz.

© 2021 Information Processing Society of Japan

```
GET /dataset/casino_0231.jpg HTTP/1.1
Host: 192.168.0.8
User-Agent: curl/7.54.0
Accept: */*
```

図 9 IoTPoCoFuzz によって最終的に生成された消費電力の大きい入力
Fig. 9 Final high power consumption output of IoTPoCoFuzz.

表 3 lighttpd に対する消費電力の増大の検証
Table 3 Verification result fore increasing power consumption on lighttpd.

機器名	最大消費電力経路値	lighttpd の消費電力 [mW]			
		初期	入力 1	入力 2	入力 3
機器 1	3.871	0.45	5.02	3.91	5.36
機器 2	2.320	0.62	5.21	4.46	4.87
機器 3	3.734	0.39	4.52	3.98	4.17

表 4 mosquitto に対する消費電力の増大の検証
Table 4 Verification result for increasing power consumption on mosquitto.

機器名	最大消費電力経路値	mosquitto の消費電力 [mW]			
		初期	入力 1	入力 2	入力 3
機器 1	0.258	1.83	4.12	3.71	4.31
機器 2	0.237	1.54	2.46	4.40	5.06
機器 3	0.313	1.34	2.46	3.64	5.08

ることが分かる．ファイルを確認したところ，図 9 でリクエストしているファイルは，図 8 と比較して，非常に大きいファイルであった．このことから，送信パケット量やファイル I/O の増大が消費電力の増大要因となっていると考えられる．

5.5 同一環境の対象機器に関する検証

5.4 節では，特定のファジング対象機器に対して消費電力が増大することを検証した．本節では，5.4 節までに使用したファジング対象機器を含めた 3 台のファジング対象機器（機器 1–3 とする）を用意し，5.3 節および 5.4 節と同様，IoTPoCoFuzz を用いて消費電力の大きい入力を発見する（発見した入力をそれぞれ入力 1–3 とする）．その後，機器 1–3 を検証対象機器として扱い，入力 1–3 を機器 1–3 へそれぞれ入力することで，特定の機器で発見した入力が，異なる機器においても消費電力が増加する入力として機能するかどうかを検証する．

機器 1–3 で IoTPoCoFuzz を用いて発見した入力 1–3 を機器 1–3 で動作する検証対象ソフトウェアに対して入力した際の消費電力値を，**表 3**，**表 4**，**表 5** に示す．それぞれの表から，すべての場合において，初期の入力の消費電力値より大きい消費電力値を記録する入力を生成できていることが分かる．以上のことから，IoTPoCoFuzz によって生成された入力は，同一環境の他機器でも有効であることが確認できた．

表 5 mosquitto_sub に対する消費電力の増大の検証
Table 5 Verification result for increasing power consumption on mosquitto_sub.

機器名	最大消費電力経路値	mosquitto_sub の消費電力 [mW]			
		初期	入力 1	入力 2	入力 3
機器 1	0.129	0.14	0.22	0.23	0.24
機器 2	0.096	0.12	0.20	0.29	0.24
機器 3	0.131	0.17	0.24	0.22	0.24

6. 議論と制限事項

6.1 議論

現実の実行時間を消費電力要因としてモデル化した消費電力モデルを用いて IoTPoCoFuzz を用いることで，消費電力の大きい入力を発見可能であることを示した．この結果から，IoTPoCoFuzz によって消費電力の大きい入力が発見可能なこと，同一環境の他機器に対しても有効であることが分かった．IoTPoCoFuzz によって，多数のプロトコルに対する自動検査，複雑なプロトコルに対する定義ファイルの自動生成などが期待できる．

本研究では，望ましくないイベントとして消費電力の増大を想定し，その発生条件に基づき定義ファイルの自動生成を行っている．一方，消費電力の増大以外にも，望ましくないイベントは今後多角化すると考えられる．そのようなイベントも自動的に発見可能な手法とするためには，それぞれのイベントの発生条件を解明し，発生条件に基づく自動生成手法を考える必要がある．また，本実験では，ソフトウェアのキャッシュを使用しない状態で実験を行ったが，現実の IoT 機器上で動作するソフトウェアでは，キャッシュを使用する場合も存在する．そのため，キャッシュを使用した場合における消費電力モデルの作成方法についても検討する必要がある．

表 3 から表 5 を比較すると，lighttpd とそれ以外の最大消費電力経路値の差が大きい．この差は，MQTT 通信がバイナリベースのプロトコルであり，トークンベースのプロトコルである HTTP 通信と比較して，プロトコルフォーマットや型の推論が困難であったため，自動生成された定義ファイルが良質でなかった結果である．

6.2 制限事項

6.2.1 未知のプロトコルへの対処

IoTPoCoFuzz では，通常のファジングとは異なり，初めに送信するリクエストに発する一連のやりとりを含めた通信によって，どの程度消費電力が増加するのか確認する．そのため，IoTPoCoFuzz は初めに送信するリクエストのみをファジングの入力として定義ファイルに沿って生成し，

© 2021 Information Processing Society of Japan

それ以降のリクエストの送信，レスポンスの対応，追加のリクエストの生成は，ライブラリや自作プログラムに任せる方針を採っている．また，定義ファイルの自動生成に関しても，同様の方針に従っている．このような方針を採ることで，ファジングによって生成された入力以外を原因とする異常な通信の発生を抑制している．この方針に従って通信を生成する際には，各通信で公開されているライブラリを移植することで，容易に生成可能である．

しかし，IoTPoCoFuzz のプロトタイプの実装では，ライブラリや自作プログラムに対応していない未知のプロトコルに対して IoTPoCoFuzz を実行することができない．未知のプロトコルに対応させるためには，PULSAR のように，複数のリクエスト・レスポンスに対応したファザーを設計する必要がある．加えて，複数のリクエストを含んだ定義ファイルを生成する場合においても，正常な通信と異常な通信の両方を適切に生成可能な定義ファイルを生成する必要がある．

6.2.2 正確な消費電力の推定

IoTPoCoFuzz では，消費電力測定機器として Extech380803 を用いている．Extech380803 は，サンプリングの更新速度が 2.5 回/秒であるため，400 ミリ秒（400,000,000 ナノ秒）に 1 度しか消費電力を測定することができない．5.3 節の消費電力モデルの作成では，永続的にシステムコールを実行するプログラムに対して消費電力を測定しているため，サンプリングの更新速度に関係なく正確に消費電力を収集・推定できる．対して，本実験における各ソフトウェアの 1 回あたりの実行時間の範囲は，21,044,969 ナノ秒から 673,915,147 ナノ秒の範囲であった．このとき，Extech380803 では 1 度サンプリングを収集するまでにプログラムの実行が終了してしまうことがほとんどであるため，5.3 節および 5.5 節の検証対象ソフトウェアの消費電力の測定では正確な消費電力の収集・推定が難しい．より正確に消費電力を推定するには，オシロスコープなど，サンプリング更新速度の高い機器を使用して消費電力を測定する必要がある．

7. おわりに

本論文では，IoTPoCoFuzz を用いて，消費電力の大きい入力を半自動で，かつ効率的に発見する手法を実現した．さらに，RaspberryPi 上で動作する複数のソフトウェアを対象として，提案手法によるファジングを行うことで，消費電力の大きい入力を効率的に発見可能であることを実験的に示した．本手法は，消費電力モデルの作成後，自動生成した定義ファイルをファジングに用いることで，効率的に消費電量の大きい入力を発見することが可能となった．これにより，省電力なアプリケーションの開発の一助となることが期待できる．

一方，未知のプロトコルへ対応した手法の実現，正確な消費電力の推定手法などが今後の課題である．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発見可能な手法を実現することを目指し，検討を進めていきたい．

参考文献

[1] Özkan, S.: Browse cve vulnerabilities by date, available from ⟨https://www.cvedetails.com/browse-by-date.php⟩ (accessed 2020-11-30).

[2] Soltan, S., Mittal, P. and Poor, H.V.: BlackIoT: IoT Botnet of High Wattage Devices Can Disrupt the Power Grid, *27th USENIX Security Symposium* (*USENIX Security 18*), pp.15–32, USENIX Association (2018).

[3] CISA: CrashOverride Malware | CISA, available from ⟨https://us-cert.cisa.gov/ncas/alerts/TA17-163A⟩ (accessed 2020-11-30).

[4] Acar, H., Alptekin, G.I., Gelas, J.-P. and Ghodous, P.: The Impact of Source Code in Software on Power Consumption, *International Journal of Electronic Business Management*, Vol.14, pp.42–52 (online), available from ⟨https://hal.archives-ouvertes.fr/hal-01496266⟩ (2016).

[5] 神山 剛，稲村 浩，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，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，Vol.2013, pp.286–292 (2013).

[6] Pereyda, J.: jtpereyda/boofuzz: Network Protocol Fuzzing for Humans, available from ⟨https://github.com/jtpereyda/boofuzz⟩ (accessed 2020-11-30).

[7] Peach Tech: Home – Peach Tech, available from ⟨https://www.peach.tech⟩ (accessed 2020-11-30).

[8] Hsu, Y., Shu, G. and Lee, D.: A model-based approach to security flaw detection of network protocol implementations, *2008 IEEE International Conference on Network Protocols*, pp.114–123 (2008).

[9] Gorbunov, S. and Rosenbloom, A.: AutoFuzz: Automated Network Protocol Fuzzing Framework, *IJCSNS International Journal of Computer Science and Network Security*, Vol.10 (2010).

[10] Gascon, H., Wressnegger, C., Yamaguchi, F., Arp, D. and Rieck, K.: Pulsar: Stateful Black-Box Fuzzing of Proprietary Network Protocols, *Security and Privacy in Communication Networks – 11th International Conference*, Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, Vol.164, pp.330–347, Springer (2015).

[11] lcamtuf: american fuzzy lop, available from ⟨http://lcamtuf.coredump.cx/afl/⟩ (accessed 2020-11-30).

[12] LLVM Project: libFuzzer – a library for coverage-guided fuzz testing. – LLVM 12 documentation, available from ⟨https://llvm.org/docs/LibFuzzer.html⟩ (accessed 2020-11-30).

[13] Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C. and Vigna, G.: Driller: Augmenting Fuzzing Through Selective Symbolic Execution, *Network and Distributed System Security Symposium* (2016).

[14] Petsios, T., Zhao, J., Keromytis, A.D. and Jana, S.: SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities, *Proc. 2017 ACM SIGSAC Conference on Computer and Communications Security*, *CCS*, pp.2155–2168, ACM (2017).

[15] Michael Sutton and Adam Greene: The Art of File For-

© 2021 Information Processing Society of Japan

mat Fuzzing, *Blackhat USA Conference* (2005).
[16] google: google/syzkaller: syzkaller is an unsupervised coverage-guided kernel fuzzer, available from ⟨https://github.com/google/syzkaller⟩ (accessed 2020-11-30).
[17] intel: PowerTOP | 01.org, available from ⟨https://01.org/powertop/⟩ (accessed 2020-11-30).
[18] Krueger, T., Gascon, H., Krämer, N. and Rieck, K.: Learning Stateful Models for Network Honeypots, *Proc. 5th ACM Workshop on Security and Artificial Intelligence*, pp.37–48 (2012).

推薦文

本論文はIoT機器の挙動のあやしさを検知する手法の1つとして，電力消費量の変化を取り上げている．提案手法であるIoTPoCoFuzzはIoT機器の消費電力のうち，比較的大きい入力をリモートから効率的に発見することができる．ネットワークを監視することによりファジングに必要な定義ファイルを半自動で生成する機能を持ち，Raspberry Pi上で動作する複数のソフトウエアに適用した結果が報告されている．IoT機器への攻撃は社会的な問題として認識されているなか，着想の新しさと検知方法のシンプルさから今後の関連研究を誘引する有意義な論文である．以上を総合し，研究会推薦に値する論文であると判断した．

（コンピュータセキュリティシンポジウム2020
プログラム委員長　森 達哉）

水野 慎太郎

2019年静岡大学情報学部情報科学科卒業．2021年同大学大学院総合科学技術研究科情報学専攻修士課程修了．在学中は情報セキュリティに関する研究に従事．

西垣 正勝　（正会員）

1990年静岡大学工学部光電機械工学科卒業．1995年同大学大学院博士課程修了．日本学術振興会特別研究員（PD）を経て，1996年静岡大学情報学部助手．同講師，助教授の後，2010年より同創造科学技術大学院教授．博士（工学）．情報セキュリティ全般，特にヒューマニクスセキュリティ，メディアセキュリティ，ネットワークセキュリティ等に関する研究に従事．2013～2014年情報処理学会コンピュータセキュリティ研究会主査，2019～2020年情報環境領域委員長，2020年調査研究運営委員長．2015～2016年電子情報通信学会バイオメトリクス研究専門委員会委員長．2016年より日本セキュリティマネジメント学会編集部会長．本会フェロー．

大木 哲史　（正会員）

2002年早稲田大学理工学部電子情報通信学科卒業．2004年同大学大学院理工学研究科電子・情報通信学専攻修士課程修了．2010年早稲田大学理工学術院情報・ネットワーク専攻博士（工学）取得．2010年早稲田大学理工学総合研究所次席研究員，2013年産業技術総合研究所特別研究員を経て，2017年より静岡大学大学院総合科学技術研究科講師，2020年同大学准教授．情報セキュリティ全般，特に個人認証を中心としたネットワークセキュリティに関する研究に従事．電子情報通信学会会員．

© 2021 Information Processing Society of Japan