AutoPoCo: 消費電力を考慮したモデルベースファジングの自動化

水野 慎太郎[1,a)]　西垣 正勝[1,b)]　大木 哲史[1,c)]

概要：本稿では，定義ファイルを自動生成することで，IoTPoCoFuzz の実行を自動化する AutoPoCo を提案する．IoTPoCoFuzz は，あらかじめ定義された動作モデルに基づき消費電力の大きい入力をリモートから発見するモデルベースファジングであり，実行には，動作モデルとしてネットワークサービスのプロトコルフォーマットを含んだ定義ファイルが必要となる．IoTPoCoFuzz では，定義ファイルを手動で作成することが前提であったが，作成に必要な専門知識やその作成コスト，さらには技術仕様等が明らかでない場合の対応が問題となっていた．そこで，AutoPoCo では，ネットワークトラフィックから定義ファイルを自動的に生成する手法を提案する．AutoPoCo は，プロトコルフォーマットの各トークン型を推論し，異常な通信だけではなく，正常な通信も行う変異手法を選択する．これにより，生成した定義ファイルを用いて，効率的に消費電力の大きい入力を発見することを可能とした．AutoPoCo のプロトタイプから生成した定義ファイルを IoTPoCoFuzz に使用することで有効性を示す．

キーワード：消費電力，自動化，プロトコル

AutoPoCo: Automation of Power Consumption-aware Model-based Fuzzing

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

Abstract: In this paper, we propose AutoFuzz, which automates IoTPoCoFuzz by automatically generating definition files. IoTPoCoFuzz is model-based fuzzing that remotely finds an input that causes significant power consumption based on a pre-defined behavioral model. The execution of IoTPoCoFuzz requires a definition file that contains a protocol format of network service. IoTPoCoCoFuzz assumes that definition files are created manually. However, there were challenges in terms of expertise required to create definition files, production costs, and dealing with unknown technical specifications. Therefore, AutoPoCo proposes that automatically make definition files from network traffic. AutoPoCo infers each token type in the protocol format and chooses a mutation that communicates not only abnormal communication but also normal communication. Thereby, we were able to efficiently find an input that causes significant power consumption using the generated definition file. Our paper shows the effectiveness of using the definition file generated from the AutoPoCo for IoTPoCoFuzz.

Keywords: Power Consumption, Automation, Protocol

1. はじめに

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・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，環境やサービスごとに多角化していくと考えられる．

1 静岡大学大学院総合科学技術研究科
Graduate School of Integrated Science and Technology, Shizuoka University
a) mizuno@sec.inf.shizuoka.ac.jp
b) nisigaki@inf.shizuoka.ac.jp
c) ohki@inf.shizuoka.ac.jp

© 2020 Information Processing Society of Japan

著者らは，望ましくないイベントのうち消費電力の増大に着目し，IoT 機器といったローカルでの入力やファザーの起動に制限のある機器上で動作しているサービスに対して，消費電力の大きい入力を発見する IoTPoCoFuzz[1] を提案した．IoTPoCoFuzz は，検証するネットワークサービスでの通信を実現する際に使用さ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記述した定義ファイルを予め手動で作成する必要がある．これらの定義ファイル，特にプロトコルフォーマットが正しいものでない場合，IoTPoCoFuzz は正確に実行されず，消費電力の大きい入力を発見することはできない．加えて，消費電力の増大は，脆弱性を目的としたファザーが多く生成する異常な入力だけではなく，正常な入力においても発生し得るため，正常と異常両者の通信を検証可能としなくてはならない．

そこで本稿では，専門知識を必要とせず，短時間でのプロトコルフォーマットの定義を可能とするため，IoTPoCoFuzz のための定義ファイルの生成から，IoTPoCoFuzz の実行までを自動化する AutoPoCo を提案する．プロトコルフォーマットの推定を伴うネットワークサービスやプロトコルに対するファジング手法はこれまでもいくつかの手法が提案されている [2], [3], [4]．これらの手法では，ネットワークトラフィックから自動的にプロトコルフォーマットや通信手順を推定している．いずれの手法も，脆弱性を特定することを目的としていることから，生成するデータの多くは異常な通信となるが，これに対し，望ましくないイベントは正常な通信中にも発生する可能性があることがわかっている [1]．したがって，IoTPoCoFuzz の目的を満たすには，型推論を行った上で適切な変異手法を選択し，多くの正常な通信データの生成を含めたプロトコルフォーマットを定義する必要がある．

AutoPoCo は，検証対象機器の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を推論する．また，推論したプロトコルフォーマットのトークン（特定のルールに従って記載された文字列）の型も推論することで，トークンごとの変異手法を適切に選択する．変異手法の選択においては，異常な通信だけではなく，正常な通信も生成可能な変異手法を選択することで，効率的に消費電力の大きい入力を発見する．本手法は，ネットワークトラフィックを監視することで，専門知識を必要とせず，短時間で検証するネットワークサービスのプロトコルフォーマットを自動的に生成することが可能である．AutoPoCo の有効性評価においては，RaspberryPi 上で動作する複数のソフトウェアに対して AutoPoCo を実行し，生成された定義ファイルを用いて IoTPoCoFuzz を実行することで，専門知識を必要とせず，効率的に消費電力の大きい入力を発見できることを示す．

本研究の貢献は，次のようにまとめられる．

- ネットワークプロトコルを解析することで，IoTPoCoFuzz の定義ファイルの作成から実行までを自動化する AutoPoCo を提案した．
- AutoPoCo を用いて生成した定義ファイルを用いて，消費電力の少ない機器，かつ現実世界で使用されているソフトウェアに対して IoTPoCoFuzz を実施することで，AutoPoCo 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テスト手法の一種である．近年では，AFL[5] や libFuzzer[6] 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，ハング，異常終了，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．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．

ネットワークを通じて，ファザーを動作させてい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に関する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値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．たとえば，Boofuzz[8] はプログラム中に，Peach[9] は XML 形式で記述する Pit ファイルに，それぞれ定義する．

定義したモデルに基づいて入力値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．ネットワークの他にも，SPIKEfile[10] のようなファイルを対象としたもの，syzkaller[11] 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 モデルの自動生成

モデルベースファジングでは，2.1 節で述べた通り，事前に定義した動作モデルに基づいたファジングを行う．ファジングの実行に際しては，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 [2] 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．推論結果を元に自動生成し

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[3] は，Hsu らと同様に，ネットワークトラフィックから状態遷移モデルやメッセージを推論した．Hsu らと異なる点として，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た．

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，異常な通信のみならず，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 IoTPoCoFuzz 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 PULSAR

AutoPoCo で行われる，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は，PULSAR[4] のモデル推論手法に基づいて行われる．PULSAR 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[12] に基づいており，その手順は以下の通りである．

(1) 通信をキャプチャしただけのネットワークトラフィックデータから，一定時間内に送受信しているパケットをメッセージとしてまとめる．この時，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みなし，セッションごとに識別子を付与する．

(2) 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 をそれぞれ次元とみなし，メッセージごとにクラスタリングを行う．PRISMA によるクラスタリングでは，各メッセージがどのクラスタに属するか判別する．クラスタリングを行う際，定数やランダムな文字列を削除するため，次元削減を行う．

(3) ネットワークプロトコルの状態遷移を模倣するため，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元に，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) 状態遷移モデルの各状態遷移におけるメッセージを生成するため，手順 1 のメッセージと手順 3 の状態遷移モデルを元に，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，トークン・n-gram の異なる箇所があった場合，そのトークン・n-gram 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，どのトークン・n-gram が可変か，といった情報を付与する．

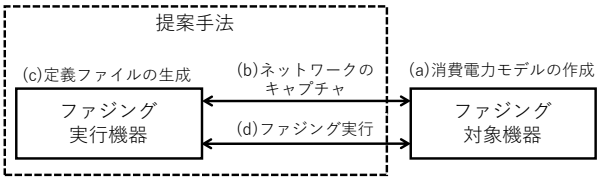

図 1 IoTPoCoFuzz と提案手法を用いた検証対象機器への検証手順

3. 提案手法

3.1 提案手法概要

IoTPoCoFuzz と AutoPoCo を用いて，検証対象機器に対して検証を行うまでの手順を図 1 に示す．AutoPoCo は，図 1 のうち，手順 (b) から手順 (d) に該当する．まず，消費電力に関連すると思われる処理に関して，処理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 (a) する．次に，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 (b) し，AutoPoCo を用いて定義ファイルを自動的に生成 (c) し，IoTPoCoFuzz によるファジングを実行 (d) する．

図 1 のうち，ネットワークトラフィックデータから定義ファイルを生成するまでの手順 (b〜d) の詳細を図 2 に示す．まず，キャプチャ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．この時，トークンごとに適切な変異手法を選択する．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通信方式を特定 (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．

3.2 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 節に記載した 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 に示す通りである．アルゴリズム 1 の各関数のうち，2–5 行目は，2.3 節の手順 1–4 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 節との差分

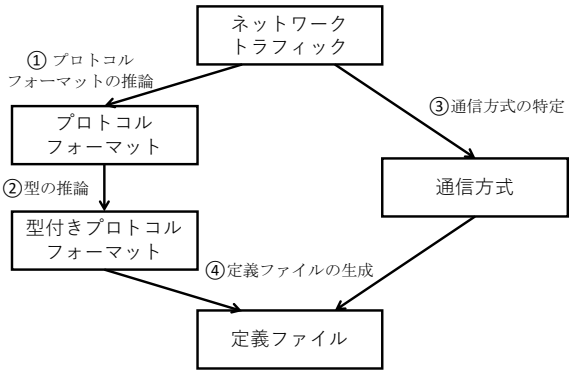

図 2　提案手法を用いた定義ファイルの自動生成手順

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
```

であり，正常な通信と異常な通信の両者を検証可能とするために，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．

3.2.1 CreateProtocolFormat

状態遷移モデルとメッセージテンプレートを元に，プロトコルフォーマットを作成する．PULSAR のモデル推論によって生成されたモデルである状態遷移モデルとメッセージテンプレートは，複数リクエスト・レスポンスを送受信するセッションに対応している．一方，IoTPoCoFuzz では，はじめに送信するリクエストを生成した後，残りの通信を別プログラムに任せることで，正常な通信を生成する確率を向上させている．そのため，プロトコルフォーマットでは，状態遷移モデルにおいて，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のみをモデル化することで，IoTPoCoFuzz においても使用可能とする．

3.3 型の推論

型の推論では，取り出したトークンの値から，数値・文字列・バイナリといったトークンの型を判別する．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．

アルゴリズム 2 に，AutoPoCo の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．

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
```

3.3.1 TokenTypeInference

TokenTypeInference では，トークンの値によって型を推論することで，そのトークンに望ましい変異手法を選択できるようにする．推論する型は，数値，文字列，バイナリに大別され，型によって SelectMutate で選択される変異手法が大きく異なる．

3.3.2 SelectMutate

TokenTypeInference で推論された型と，トークンの値を合わせて，トークンに適切な変異手法を選択する．まず，推論された型によって，型に合わせた変異手法を選択する．次に，トークンの値を参考に，トークンに適していると思われる変異手法を追加する．たとえば，文字列の場合，ファイル名，ホスト名，ディレクトリパスなど，トークンの値に類似している文字列関係の変異手法を追加する．

4. 実装

AutoPoCo のプロトタイプの実装にあたり，モデルの推論は PULSAR[*1]および自作の Python プログラム，型の推論は自作の Python プログラムを使用した．モデルの推論の実装では，PULSAR によって生成された状態遷移モデルとメッセージテンプレートを元に，状態遷移モデルにおいて，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のみをモデル化する．型の推論では，IoTPoCoFuzz のプ

[*1] https://github.com/hgascon/pulsar

ロトタイプの実装である Peach[*2]に定義されている型および変異手法を中心に，型の推論や変異手法の選択を行う．なお，IoTPoCoFuzz 独自の変異手法として，特定のディレクトリ内に存在するファイル名を一つ返す変異に関しても，変異手法選択時の対象とする．

5. 実験・検証

本実験は，以下の 2 点の実験・検証を通して，AutoPoCo の信頼性と有用性を示すことを目的とする．

- 自動および手動で生成した定義ファイルを利用した場合の IoTPoCoFuzz の実行結果を比較することで，AutoPoco が自動で生成した定義ファイルの信頼性を確認する．
- AutoPoCo および 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，提案手法の有用性を確認する．

ここでは，IoT 機器で使用される代表的なプロトコルの例として，HTTP と 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，MQTT については軽量なメッセージプロトコルである MQTT のメッセージを中継するサーバの役割を持つ mosquitto を対象ソフトウェアとして用いる．

5.1 実験環境

全ての実験・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 16GB 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．ファジング実行機器からファジング対象機器に対して遠隔でファジングを実行するファザーとして IoTPoCoFuzz を使用し，さらに 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ものを使用した．また，消費電力測定機器として Extech 380803[*3]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，PowerTOP バージョン 2.10，lighttpd，mosquitto を使用した．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．

5.2 実験準備

5.2.1 ネットワークトラフィック

AutoPoCo による定義ファイルの自動生成には，キャプチャしたネットワークトラフィックが必要となる．lighttpd，mosquitto 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．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 lighttpd では，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[*4]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[*5]から無作為に 5,000 ファイルを収集した．

5.2.3 消費電力モデルの作成

ファジング対象機器上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．記録した消費電力から文献 [1] 式 (1) に基づき消費電力値を計算し，消費電力モデルに追加した．この時，0mW といった極端に小さな値は定常雑音とみなして計算に用いないこととした．なお，本提案手法は定常状態よりも消費電力が大きくなる入力を効率的に発見することを目的としている．したがって，全ての消費電力要因の動作を行うプログラムに使用している usleep() 使用時の電力を定常状態と仮定し，usleep() 使用時の消費電力からの増分比率を消費電力値として用いた．これにより，環境条件によって異なる定常状態の消費電力による影響を低減させることができる．

また，バイト数を扱う消費電力要因に関しては，文献 [1] に従い，上限を 1,024KB として，バイト数による消費電力の変化を考慮することとした．

5.3 自動生成した定義ファイルの信頼性検証

ファジング対象機器上で動作する lighttpd，mosquitto のそれぞれに対して，手動と自動で作成した定義ファイルを使用した，IoTPoCoFuzz を用いたファジングを実施した．図 3，図 4 に，ファジング対象機器に対して，lighttpd，mosquitto のそれぞれに，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高消費電力経路値（文献 [1] 式（2））の関係を示す．それぞれの図に対する結果から，自動生成した定義ファイルを用いた場合においても，初期の入力よりも大きい最大消費電力経路値を発見している．また，図 3 では，自動生成した定義ファイルを用いて手動と同等の最大消費電力経路値を発見しており，手動と比較すると劣るものの，自動においても効率的に消費電力経路を発見できることが確認できた．

*2 https://github.com/MozillaSecurity/peach
*3 http://www.extech.com/products/380803
*4 https://www.mediawiki.org/wiki/API:Main_page/ja
*5 http://web.mit.edu/torralba/www/indoor.html

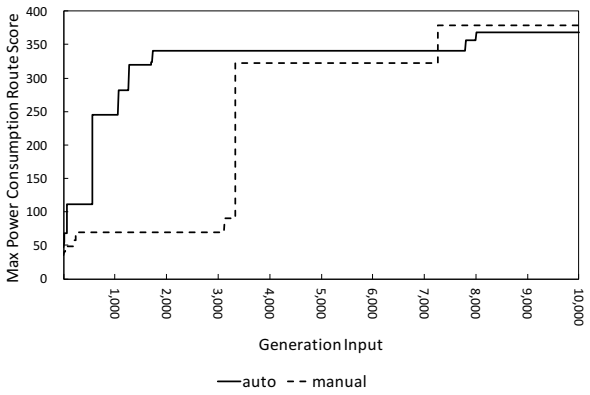

図 3　lighttpd に対する変異回数ごとの最高消費電力経路値

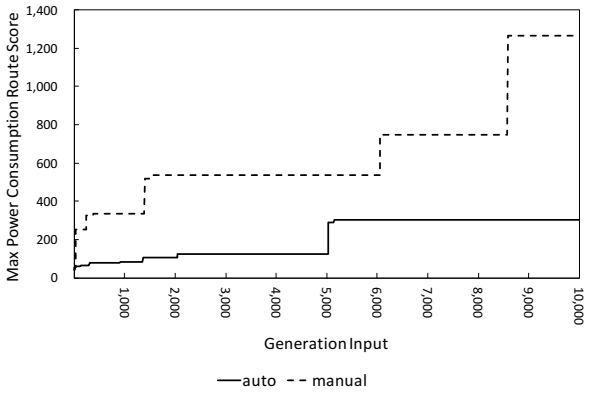

図 4　mosquitto に対する変異回数ごとの最高消費電力経路値

5.4　同一環境の対象機器に関する検証

5.3 節では，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，5.3 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）を用意し，5.3 節と同様，IoTPoCoFuzz を用いて消費電力の大きい入力を発見する．その後，機器 1–3 を検証対象機器として扱い，各自発見した消費電力の大きい入力を選択することで，消費電力が増加するか検証する．

機器 1–3 で IoTPoCoFuzz を実行し，発見した最大消費電力経路値，最大消費電力経路値の入力をファジング対象機器で動作する lighttpd，mosquitto に対して入力した際の消費電力値を，それぞれ表 1，表 1 に示す．それぞれの表から，手動と自動の両方において，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．また，表 1 では，複数の状況において，自動の消費電力値が手動の消費電力値を上回っている．以上のことから，AutoPoCo および 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．

6.　議論と制限事項

6.1　議論

自動的に生成した定義ファイルを用いて，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，AutoPoCo を用いた IoTPoCoFuzz の自動化が有効であること，発見した消費電力の大きい入力が同一環境の他機器に対しても有効であることがわかった．AutoPoCo によって，多数のプロトコルに対する IoTPoCoFuzz の自動検査，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．今後，より一般的に同一環境で動作する他機器に対する有効性を示すためには，検証に使用する台数，およびソフトウェアの種類を増やしていく必要がある．

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だろう．

AutoPoCo のプロトタイプでは，定義ファイルを自動生成する際にネットワークトラフィックのみを参照するため，技術仕様を読みながら手動で作成された定義ファイルと比較して劣ったものとなる．結果として，機器 2 の lighttpd を除いて，手動の最大消費電力経路値が自動よりも大きい値となった．また，表 1 と表 2 を比較すると，mosquitto の手動と自動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，正常な通信を多くする工夫として，はじめに送信するリクエスト以外は別プログラムに任せる方針を採っており，AutoPoCo もその方針に合わせた定義ファイルの自動生成を行っている．そのため，IoTPoCoFuzz のプロトタイプの実装では，別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　トラフィック内容による依存

AutoPoCo では，ネットワークトラフィックを元に，プロトコルフォーマットやトークンの型を推論する．そのため，生成される定義ファイルは，ネットワークトラフィックに大きく依存する．ネットワークトラフィックの質が悪く異常な通信が大量に存在する場合，生成される定義ファ

表 1　lighttpd に対する消費電力の増大の検証

機器名	手動/自動	最大消費電力経路値	lighttpd の消費電力 [mW]			
			初期	機器 1	機器 2	機器 3
機器 1	手動	379.37	0.36	1.60	1.58	1.40
	自動	368.85	0.48	1.63	1.78	1.60
機器 2	手動	499.82	0.47	1.27	1.41	1.29
	自動	332.85	0.36	1.46	1.29	1.31
機器 3	手動	290.46	0.42	1.24	1.41	1.34
	自動	340.07	0.48	1.27	1.41	1.48

表 2　mosquitto に対する消費電力の増大の検証

機器名	手動/自動	最大消費電力経路値	mosquitto の消費電力 [mW]			
			初期	機器 1	機器 2	機器 3
機器 1	手動	1265.95	1.48	5.28	5.30	4.10
	自動	301.49	1.48	3.41	2.77	1.79
機器 2	手動	775.13	1.36	4.90	4.49	3.44
	自動	173.09	1.28	2.86	3.43	1.61
機器 3	手動	557.52	1.23	5.53	4.10	6.71
	自動	114.26	1.42	2.90	2.77	1.93

イルも異常な通信が大量に発生するものである可能性が高い．正常な通信と異常な通信の両方が混ざる定義ファイルを生成するためには，ネットワークトラフィックにおいても正常な通信と異常な通信の両方を含んだものである必要がある．

7. おわりに

本稿では，専門知識を必要とせず，短時間で定義ファイルを自動的に生成することで，IoTPoCoFuzz の実行を自動化する AutoPoCo を提案した．AutoPoCo では，消費電力の大きい入力を発見するための定義ファイルを生成するためにプロトコルフォーマットや型の推論，適切な変異手法の選択を行うことで，短時間で効率的に消費電力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．

提案手法について，RaspberryPi 上で動作する複数のソフトウェアを対象として，AutoPoCo を用いて自動生成した定義ファイルと，手動で作成した定義ファイルを用いて IoTPoCoFuzz によるファジングを行うことで，自動生成した定義ファイルにおいても，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．

一方，未知のプロトコルへ対応した定義ファイルの生成，ネットワークトラフィックによる定義ファイルの質など，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可能な定義ファイルの自動生成ができるよう，改良を行っていく．

参考文献

[1] 水野慎太郎，西垣正勝，大木哲史：IoTPoCoFuzz: IoT 機器に対する消費電力を考慮したファジング，ハードウェアセキュリティ研究会 (2020).

[2] 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).

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

[4] 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, Springer, pp. 330–347 (2015).

[5] lcamtuf: american fuzzy lop, `http://lcamtuf.coredump.cx/afl/` (参照日: 2020/8/20).

[6] LLVM Project: libFuzzer – a library for coverage-guided fuzz testing. — LLVM 12 documentation, `https://llvm.org/docs/LibFuzzer.html` (参照日: 2020/8/20).

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

[8] Joshua Pereyda: jtpereyda/boofuzz: Network Protocol Fuzzing for Humans —boofuzz 0.1.6 documentation, `https://github.com/jtpereyda/boofuzz` (参照日: 2020/8/20).

[9] Peach Tech: Home - Peach Tech, `https://www.peach.tech` (参照日: 2020/8/20).

[10] Michael Sutton and Adam Greene: The Art of File Format Fuzzing, *Blackhat USA Conference* (2005).

[11] google: google/syzkaller: syzkaller is an unsupervised

coverage-guided kernel fuzzer, `https://github.com/google/syzkaller` (参照日: 2020/8/20).

[12] Krueger, T., Gascon, H., Krämer, N. and Rieck, K.: Learning Stateful Models for Network Honeypots, *Proceedings of the 5th ACM Workshop on Security and Artificial Intelligence*, p. 37–48 (2012).