IoTの不具合に対するファジングツールPeachの有効性調査

永原 渓太郎[1,a)]　大野 堅太郎[1,b)]　吉田 則裕[1,c)]　高田 広章[1,d)]

概要：IoT システムにおけるセキュリティ脆弱性を素早く発見するため，ファジングツールが利用されている．本研究では，実際の IoT システムにファジングツール Peach を適用し，その有効性や課題を調査する．オープンソースとして公開されている不具合を含む 2 つの IoT システムに対して，Peach を用いてファジングを行った．その結果, 両方のプロジェクトでバグを発見できた．その一方で，4 つの課題が明らかとなった．

An Investigation of the Effectiveness of the Fuzzing Tool Peach for IoT Failures

Keitaro Nagahara[1,a)]　Kentaro Ohno[1,b)]　Norihiro Yoshida[1,c)]　Hiroaki Takada[1,d)]

1. はじめに

IoT (Internet of Things) とは，社会に存在する様々なモノが IoT 技術が通信機能を持ち，インターネットに接続したり相互に通信することができるようになることである [1]．IoT は現在，急速な発展を遂げており，現在の成長率が続けば，2025 年には，世界に 750 億台もの IoT デバイスが存在することになる [2]．これらの IoT デバイスは，産業や生活を豊かにする一方で，IoT の脆弱性を起因としたネットワーク経由の攻撃のリスクが高まっている．実際に 2016 年には，数千台の家庭用 IoT デバイスにマルウェアが感染し，DDoS 攻撃に利用された [3]. このような IoT システムへの攻撃の原因として，IoT システム特有の以下の問題によってセキュリティ対策が困難であったことが考えられる．

- リソースが限定的で対策が困難 [4].
- 市場に早く投入しようとするため，システムの入れ替わりが早く対策時間の創出が困難 [5].
- 頻繁なアップデートを行うことが困難 [6].

従って，これらの問題を解決するため，システム開発中の段階でできるだけ多くのセキュリティ脆弱性を素早く発見することが求められる [4].

ファジングはこの IoT における問題を解決しうるテスト手法であると考えられる．ファジングとはファズと呼ばれるデータを生成し，テスト対象に入力としてファズを繰り返し送信することで意図的にクラッシュを誘発させ，その原因となる脆弱性を発見するテスト手法である．ファジング導入の利点として単純さ，導入が容易である点, 様々なソフトウェアに適用できる点がある [7]. これらのファジング導入のメリットを鑑みると，IoT システムに対してファジングを活用できれば，IoT システム特有の問題を解決できると考えられる．そこで本研究では，IoT システムにファジングを行うことで，ファジングの有効性, そして実際に適用するにあたっての課題を調査する．

IoT システムに対するファジングが有効であると考えられる一方で IoT システムに対するファジングの利用はそれほど広まっていない．この利用が進まない主な理由は，利用可能なリソース不足と IoT システムにおける応答の不足にある [4]. これらの問題は IoT が組込みシステム上で実行されることに起因する．組込みシステムに対するファジン

[1] 名古屋大学
[a)] keita0515@ertl.jp
[b)] k_ohno@ertl.jp
[c)] yoshida@ertl.jp
[d)] hiro@ertl.jp

グの問題点として以下が挙げられる [4].

- 不具合の種類を検出する能力が一定でない.
- ユーザーインタフェースや対話するためのシェルがない.
- OS のサポートがない場合がある.

これらの問題により，IoT システムに対して利用できるファジングツールは限定されてしまう．Eceiza ら論文 [4] では，組込みシステムに対してファジングできるファジングツールは，DrE，FIE，IoTFuzzer[8]，Peach，S2E の 5 つ挙げられている．

これまでの研究の多くでは，ファジングでの効率的な変異手法の研究といったファジング手法の研究が多く，IoT システムに着目してファジングについて論じた研究は少ない．また IoT システムにおけるファジングについて述べたものでも，ファジングにおける IoT の特有の問題を述べたもの [4] や IoT システムに適した変異手法の提案 [5][8] といったものしかなく，特定のファジングツールでの IoT に対するファジングの有効性を論じたものはなかった．したがって本研究では，先に述べた 5 つのファジングツールの中からオープンソースであり，過去の利用実績が高いファジングツールである Peach を用いて実際の IoT システムにファジングすることで Peach の有効性を調査する．本研究の貢献は次の 2 つである．

- IoT システムに対するファジングに Peach の利用が有効であることがわかった.
- IoT システムに対してファジングツール Peach を適用するにあたっての課題を 4 つに分類した.

本論文は，本章も含めて 5 章で構成される．2 章では，本調査で利用するファジングツールである Peach について解説する．3 章では，本研究の流れを述べた後，事例 1，事例 2 での不具合の解説，ファジングの実装方法を順に述べる．4 章では，事例 1，事例 2 のそれぞれの調査結果を述べる．5 章では，調査結果からの考察を述べ，考察内で挙げた課題を分類する．最後に 6 章では，本研究のまとめ，今後の課題について述べる．

2. ファジングツール Peach

Peach[*1] とは Peach Fuzzer 社が開発したソフトウェアで，複数の OS 上で動作し，ファイルを読み込むソフトウェアや TCP/IP などで通信するソフト，ウェブアプリケーションなど，幅広いソフトウェア製品に対してファジングを実践できる．Peach は，生成ベースと突然変異ベースの両方のファジングを実行できるブラックボックスファザー [7] であり，pits ファイルと呼ばれる xml ファイルに諸設定を書き込むことであらゆるソフトウェアに対応できるようになっている．このように Peach はプロトコルに基づくファザーであり，入力形式を指定できない AFL 等のファザーとは異なる．Peach では，ファジングの設定を pits ファイルと呼ばれる xml ファイルで行う．pits ファイルには初期シード，通信の遷移，テストインターフェース，変異戦略を記述する．初期シードはブロックごとに記述できる．標準で用意されているランダム戦略では，予めいくつかの変異戦略が用意されており，はじめに pits ファイル上の初期シードに対してどの変異戦略が適用できるか判断する．その後，ファズ送信ごとに初期シードから選ばれた遷移戦略をブロックごとにランダムに適用することで遷移していく．またテストインターフェースは Publisher と呼ばれ，通信プロトコルに合わせていくつか用意されている．また用意されているもの以外のプロトコルにファジングを行うときは，Publisher は自分で記述する必要がある．Publisher はファジング手順に合わせて対応する関数が用意されており，その関数内を記述することで Publisher を作成することができる．

Peach には 2 系と 3 系が存在し，以下では 2 系についてのべる．Peach には，設定ファイルとして pits ファイルと呼ばれる xml 形式のファイルが用意されており，適切に記述する必要がある．pits ファイルには，いくつか要素があり，XML ヘッダ，Include 要素，DataModel 要素，StateModel 要素，Agents 要素，Test 要素，Run 要素の 7 種類の要素を主に持つ．要素ごとの解説を以下に示す．

Include 要素
Include 要素では，要素内で指定された別ファイルを読み込む．指定されたファイルは主に xml 形式で記述され，パスの追加，パッケージやライブラリの動的なインポート，使用する変異手法の定義を記述する必要がある．

DataModel 要素
DataModel 要素では，ファズの初期値の構造を定義する．テストデータの領域ごとにデータ型やサイズ，変異手法を定義できる．

StateModel 要素
StateModel 要素では，テスト対象の遷移を定義できる．State 要素とよばれる要素内で Action 要素とよばれるテスト対象との送受信の動作を指定する．

Agents 要素
Agents 要素では，メモリ消費の監視や障害の検出といった独立したプロセスを指定できる．

Test 要素
Test 要素では，いくつかの要素を用いてファジングのテスト環境を定義する．ここでは追加/除外する変異手法，ファジングで使用 StateModel 要素の定義，通信方式の定義，変異戦略の設定を記述できる．変異戦略は標準で生成ベースと突然変異ベースが用意されて

[*1] https://peachtech.gitlab.io/peach-fuzzer-community/

ソースコード 1　pits ファイルの例

```
<?xml version="1.0" encoding="utf-8"?>
<Peach xmlns="" version="1.0"
       author="" description="Hello␣World␣Example">
       <Include ns="default" src="file:defaults.xml"
            />
       <DataModel name="HelloWorldTemplate">
              <String value="Hello␣World!" />
       </DataModel>
       <StateModel name="State" initialState="State1"
            >
              <State name="State1" >
                     <Action type="output" >
                         <DataModel ref="
                              HelloWorldTemplate"/>
                     </Action>
              </State>
       </StateModel>
       <Test name="HelloWorldTest">
              <StateModel ref="State"/>
              <Publisher class="stdout.Stdout" />
       </Test>
       <Run name="DefaultRun" description="Stdout␣
            HelloWorld␣Run">
              <Test ref="HelloWorldTest" />
       </Run>
</Peach>
```

いる．通信方式も標準でいくつか用意されている．

Run 要素

Run 要素では，ファジングを行うための環境を記述する．必要であればログを取得する環境も定義できる．

ソースコード 1[*2]は pits ファイルの例である．Peach では，対応している通信形式ならば基本的に上記のような pits ファイルの編集のみでファジングを行えるようになっている．

3. 調査

3.1 調査環境

本調査は，macOS Monterey Version 12.1 上に VirtualBox を用いて，Ubuntu Desktop 20.04.3 LTS の仮想環境を構築し，仮想環境上でテストを行っている．また用いる Peach は Peach2 系である脚注[*3]を使う．

3.2 調査の流れ

まずオープンソース上の IoT システムでの不具合を持ったリポジトリを収集する．データセットは後述する Makhshari らの研究内のデータセット[*4]に発表後に発生した不具合を追加したものを使う．このデータセット上のリポジトリから GitHub 上の Issues で報告されている不具合

[*2] https://peachtech.gitlab.io/peach-fuzzer-community/v2/HowDoI.html から一部改変して引用
[*3] https://github.com/MozillaSecurity/peach
[*4] https://github.com/IoTSEstudy/IoTbugschallenges

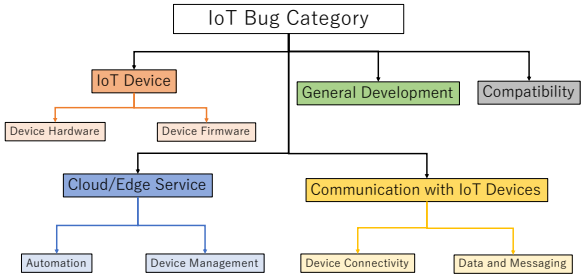

図 1　IoT システムのバグカテゴリー [9]

を見て，「クラッシュするか」，「入力に対して発生する不具合か」，「再現可能か」の 3 項目より調査対象となる事例を選出する．以上の 3 項目を選出項目としたのは理論上ファジングで発見できるものを選出するためである．具体的な選出方法として，Issues の本文上で「crash」という文字が含まれ，データセットにおけるバグカテゴリーが「Data and Messaging」に含まれるものの中から手動で再現可能か調べ，選出した．そして選出した事例に Peach を用いたファジングで不具合を発見可能か調査する．その後，同一プロジェクトより他の再現可能なレポジトリを同様の基準で選出し，同様に調査する．最後にファジングにあたっての効果の評価し，課題を精査し，分類する．

3.3 Makhshari らの研究

Makhshari らの研究 [9] は，は IoT に関わるプロジェクトのバグレポートを収集し，不具合，根本原因を元に不具合を分類したものである．データセットの収集方法を以下に示す．

(1)　GitHub 上のレポジトリからトピック機能を用いて IoT 関連のレポジトリを収集する．

(2)　レポジトリの対する評価である星が 10 個以下のものを除外する．

(3)　GitHub の Issues 上から不具合に関わるものを取得する．

このように集めたデータセットを不具合の分類と根本原因の 2 軸に分類している．図 1 に不具合の分類を示す．

3.4 事例 1：MQTT における不具合

事例 1 で対象となるレポジトリ[*5]は，ESP32 と呼ばれるマイクロコントローラの統合開発環境である．不具合はバージョン 3 以下で発生し，具体的には開発環境内の標準コンポーネントである MQTT 通信を可能とするコンポーネントで発生した．本調査では Version3.1.7 を用いた．MQTT 通信とは，TCP/IP 上で動作するコネクション指向の通信プロトコルで，publish/subscribe モデルの採用により双方

[*5] https://github.com/espressif/esp-mqtt

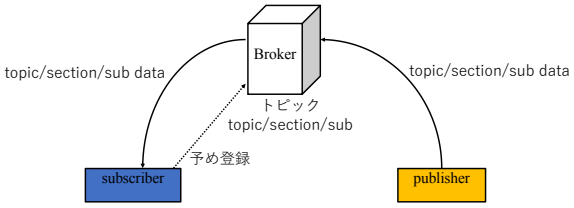

図 2 MQTT 通信のモデル

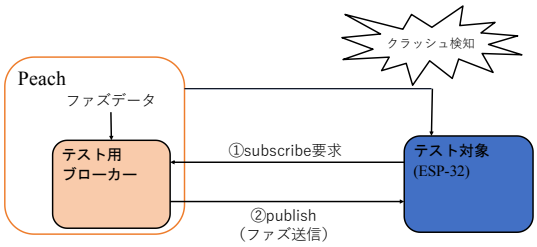

図 3 事例 1 でのテストモデル

向通信かつ低サイズを可能としており，IoT システムに適した通信プロトコルである．図 2 は MQTT 通信のモデルである．MQTT 通信では，Publisher，Subscriber，Broker と呼ばれる構成要素によって行われ，データはトピックと呼ばれるものにより分類される．トピックはスラッシュを用いて記述され *section/subsection/subsubsection* のように階層構造で記述する．MQTT 通信の一連の通信プロセスは以下のようになる．

(1) Subscriber は Broker に取得したトピックを要求する．

(2) (1) に対して Broker は Subscriber とトピックの対応を保存する．

(3) Publisher はデータにトピックをつけて Broker に送信する．

(4) (3) に対して Broker はトピックを元に予め登録してあった Subscriber にデータを送信する．

今回対象となる不具合は，Subscriber が Broker からデータを受け取るときに発生したものである．ESP32 を Subscriber としたとき，Broker から 1009bytes を超えるデータを送信したとき，クラッシュが発生する．

実装方法について解説する．図 3 は今回実装するテスト構造である．まずテスト対象は Peach 上に作られたテスト用 Broker にトピックの要求を行う．この要求は 1 回のファジングテストに対して最初の 1 回だけ行う．その後テスト対象に対してファズを複数送信していく．また変異させるテストデータ構造は，トピック部とメッセージ部に対して行うこととする．

Peach には，MQTT 通信を可能とする Publisher が用意されていない．したがって，必要な手順は Publisher の作成と pits ファイルの作成である．

ソースコード 2 事例 1 の DataModel 要素

```
<DataModel name="PublishCommand">
    <String name="topic" value="/topic/qos0"/>
    <String name="message" value="0"/>
  </DataModel>
```

また Peach 内で MQTT 通信通信を可能とするため，Peach 環境で使える MQTT スタックが必要となる．テスト実行時に必要となる仕様を以下に示す．

(1) Subscriber からの要求の取得

(2) コネクションの確立

(3) コネクションの確認に対する応答

(4) Subsriber に対するデータの送信

(5) コネクションの終了

以上の仕様を満たすテスト用 Broker を自作する．

次に作成したテスト用 Broker を用いて，Publisher を作成する．章 2 で述べたように Peach では，ファジング実行時に特定のタイミングに呼び出される関数がいくつか用意されている．したがって，関数内を記述することで Publisher を作成していく．

MQTT 通信では，コネクション指向のプロトコルであるため，通信前にコネクションを確立する．また Subscriber は予め Broker に対してトピックの要求をする必要があるため，ファジング実行前に Subscriber から要求を取得する必要がある．したがって *Initialize()* と呼ばれる最初のテスト実行時にのみ呼び出される関数に，コネクションの確立とテスト対象からの要求を取得するように記述する．

ファズの送信は *send()* と呼ばれる関数で実行される．この関数には，引数にファズが与えられており，このファズをテスト対象に送信するよう記述する．

次に pits ファイルを作成していく．pits ファイルでは，DataModel 要素，StateModel 要素，Test 要素を記述していく．

DataModel 要素では，トピック部と実際のデータを扱うメッセージ部に分ける．それぞれの初期値としてはソースコード 2 のようにトピックに */topic/qos0*，メッセージに 0 を与える．StateModel 要素では，通信の状態遷移を記述するが，今回のテストでは一連の通信ではファズの送信しか行わないため，ソースコード 3 のように Action 要素に output と呼ばれるファズの送信を行う動作を指定する．

Test 要素では，ソースコード 4 のように作成した Publisher の指定，ファズの変異方法の指定を行う．Publisher の指定では，引数としてテスト対象の IP アドレスとポートを指定する．またファズの変異方法はランダム変異を指定する．

ソースコード 3　事例 1 の StateModel 要素

```
<StateModel name="HttpRequest" initialState="
    SendRequest">
      <State name="SendRequest">
        <Action type="output">
          <DataModel ref="HttpRequestHeader"/>
        </Action>
      </State>
  </StateModel>
```

ソースコード 4　事例 1 の Test 要素

```
<Test name="DefaultTest">
      <StateModel ref="HttpRequest"/>
      <Publisher class="tcp.Tcp">
        <Param name="host" value="192.168.0.192"/>
        <Param name="port" value="8000"/>
      </Publisher>
      <Strategy class="rand.RandomMutationStrategy"
          params="MaxFieldsToMutate=4"/>
  </Test>
```

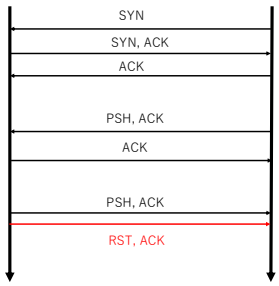

図 4　事例 2 での一連のプロセス

3.5　事例 2：TCP ヘッダにおける不具合

事例 2 で対象となるレポジトリ[*6]は，ArduinoIDE で ESP32 を利用するためのパッケージである．このパッケージは espressif 公式が発表したものである．本調査では Version1.0 を用いる．不具合の内容は TCP 通信の一連の通信プロセスにおいて正常の通信では PSH パケット受信後 FIN パケットが受信するが，PSH パケット受信後すぐに RST パケットを受信するとクライアント側でクラッシュが発生するというものである．これはパッケージ内の TCP スタック上で起きた不具合である．PSH パケットを受信すると TCP スタックは上位アプリケーションにペイロード部を渡すが，その処理途中にコネクションの強制終了を示す RST パケットを受信すると強制的に受信データのバッファが削除されるため，PSH パケット処理時のバッファ参照でエラーが起き，クラッシュする．図 4 は今回の通信の流れを示したものである．図中の赤線部でクラッシュが発生する．実装方法について解説する．今回のテスト仕様は以下である．

- PSH パケット送信後の FIN パケットをファジングする
- TCP ヘッダに対してファジングする

この仕様を満たすようにファジングを設計する．また Peach では，テスト対象からの入力に起因するテストデータの値の変化を行うことができない．したがって今回のテストでは，対象となる領域であるシークエンス番号，確認応答番号，送信元ポートをファジング対象から除外する．Peach で標準で対応している Publisher は TCP/IP 上のプロトコルである．今回の不具合は TCP レベルで発生した不具合であり，TCP ヘッダレベルでファジングを行う必要がある．したがって TCP ヘッダレベルで扱える Publisher を作成する必要がある．一般に TCP レベルでの処理は，OS の TCP スタックを用いて行うが，今回のテストでは TCP より下のレベルでのデータを扱う必要があるため，OS の TCP スタックを用いる事ができない．したがって，デバイスドライバから直接データを取得し，OS の上で動作するテスト用 TCP スタックを作成する．またファジング実行時に OS 上の TCP スタックが干渉するのを防ぐため，iptables を用いて OS の TCP スタックを遮断しておく．

OS の TCP スタックの遮断

```
sudo iptables -t raw -A PREROUTING -p tcp –dport
80 -j DROP
```

次にこのテスト用 TCP スタックを用いて Publisher を作成する．章 2 で述べたように Peach では，ファジング実行時に特定のタイミングに呼び出される関数がいくつか用意されている．したがって関数内を記述することで Publisher を作成していく．今回のテストで用いる関数は *start()*，*accept()*，*send()*，*receive()* である．*start()* は一連のテスト実行毎の最初に呼び出される．ここでは，ソケットの準備を記述する．*accept()* は通信の確立時に呼び出される関数であり，ここでテスト対象からの通信の受け入れるようにする．*send()* は実際にファズを送信する関数でここで取得したファズを送信する．*receive()* はテスト対象からのデータ受信時に実行し，受信データを返す関数である．今回のテストでは一連の通信プロセスの中でテスト対象からデータを受信する必要があるため，この関数内にデータを取得するように記述する．

次に pits ファイルの作成を行う．pits ファイルでは，DataModel 要素，StateModel 要素，Test 要素を記述していく．DataModel 要素では，ファジング元となるデータ構造とデータ取得時のデータ構造を記述する．

StateModel 要素では，テストプロセスを記述していく．ソースコード 5 は StateModel 要素を記述したものである．まず通信の確立を行うため，Action 要素に accept を指定

[*6] https://github.com/espressif/arduino-esp32

ソースコード 5 事例 2 の StateModel 要素

```
<StateModel name="TcpModel" initialState="acception">
    <State name="acception">
      <Action type="accept"/>
      <Action type="input">
        <DataModel ref="RecvCommand"/>
      </Action>
      <Action type="output">
        <DataModel ref="PublishCommand3"/>
      </Action>
    </State>
  </StateModel>
```

ソースコード 6 事例 2 の Test 要素

```
<Test name="Test">
    <StateModel ref="TcpModel"/>
    <Publisher class="my_publish.MyTcp">
      <Param name="port" value="80"/>
      <Param name="interface" value="enp0s3"/>
      <Param name="host" value="192.168.0.141"/>
    </Publisher>
    <Strategy class="rand.RandomMutationStrategy" />
  </Test>
```

表 1 事例 2 でのテストパターン

	初期値	文法規則
パターン 1	あり	あり
パターン 2	なし	あり
パターン 3	なし	なし

する．ここでは Publisher 上の *accept()* が実行される．次にテスト対象からのデータを受信するため，Action 要素に input を指定する．input では受信したデータを指定した DataModel に従って解析するため，DataModel 要素で記述したデータ取得時用のデータ構造を指定する．最後にテスト対象にファズを送信するために Action 要素に output を指定する．次にテスト要素を記述する．ソースコード 6 は Test 要素を記述したものである．ここでは記述した StateModel の指定，作成した Publisher の指定，変異戦略の指定を行う．作成した Publisher の指定では Publisher の引数も記述する．今回のテストでは，ポート，使用するインターフェース，テスト対象の IP アドレスを記述する．変異戦略は Peach に標準で用意されているランダム戦略を用いる．実際のテストでは，テストデータのデータ構造を変えてファジングを行う．表 1 はデータ構造のパターンを表した表であり，パターン 1 では初期値としてプロトコルに従ったデータと文法規則を与え，パターン 2 では文法規則のみ与え，パターン 3 では初期値も文法規則も与えずに行った．

4. 調査結果

事例 1，事例 2 ともにファジングによってクラッシュが確認できた．表 2 は，事例 1，事例 2 それぞれで 1 回のクラッシュに対して要したファジングの平均回数である．事例 1 では，テストを 25 回行ったうち，平均で 4.6 回のファジングでクラッシュが発見された．事例 2 では，パターン 1 では，3584 回のファジングで 200 回のクラッシュ，平均で 18 回に 1 回のクラッシュが確認され，パターン 2 では 69120 回のファジングで 4 回のクラッシュ，平均で 17280 回に 1 回のクラッシュが確認され，パターン 3 では 38912 回のファジングでクラッシュが確認されなかった．パターン 1 でクラッシュを起こしたファズの一つから今回変異させた領域のみ抜き出し，各領域ごとにわかるように書いたものは表 3 である．コントロールフラグ部を見ると RST フラッグが入っていることがわかる．同様に他のファズもコントロールフラグを見るとすべてのファズで RST フラグを持っていた．したがって対象とする不具合がファジングで発見されたことがわかる．一方でファジング実装にあたっていくつかの問題点も明らかとなった．その問題点は以下の 2 つである．

表 2 1 クラッシュに対する平均ファジング回数

事例 1		4.16 回
事例 2	パターン 1	18 回
	パターン 2	69120 回
	パターン 3	発見できず

(1) 通信ごとに異なる初期値を設定できない．

(2) ファジング設定が汎用的でない．

(1) は今回送信元ポート，シーケンス番号，確認応答番号をファジング対象に入れると，ファジングできなかったためである．これらの領域はテスト対象の入力によって値が決定する．そのためファズ送信ごとに有効な値は異なる．このような値をファジング対象にいれるとファジング効率が著しく下がってしまう．(2) は今回の不具合がタイミングに起因する不具合であり，それを発見するようなファジング設計に意図的にしているためである．今回の通信ではファジングできるポイントが複数設定できる．また Peach には 1 回のファジングで複数ファジングポイントを設定することは可能である．しかし今回のファジングで複数のファジング対象を設定すると，不具合の条件である PSH パケット受信後すぐに RST パケットを取得するという条件を達成できない．したがって今回のテスト対象では汎用

表 3 パターン 1 でのファズの領域

ソースポート	80
オフセット	20
予約	0b000011
コントロールフラグ	URG, ACK, PSH, RST, SYN, FIN
ウィンドウサイズ	0x11ff
緊急ポインタ	0xff00
ペイロード	00

的なテスト設計では不具合を発見できなかった．

5. 考察

事例 1 と 2 に共通する結果，事例 1 の結果，事例 2 の結果の順で考察を行う．その後，考察から得られた課題の分類を行い，分類ごとの考察を行う．

5.1 調査結果からの考察

事例 1，事例 2 ともにファジングによるクラッシュを確認でき，IoT システムに対して一定のファジングの有効性がわかる．一方でテストデータのデータ構造によって，クラッシュの発見効率は大きく異なることがわかった．したがってファジング設定において初期値の与え方はファジング設計において重要だとわかる．事例 2 の場合，初期値と文法規則与えた場合が最も効率よくクラッシュを発見できた．これは Peach のランダム手法において一回の変異ごとに初期値から文法規則ごとにランダムに変異するためであると考えられる．初期値を与えない場合，変異元は文法規則に従ったランダムデータとなるが，初期値を与えた場合，必ず有効なデータが変異元となる．これにより厳しいプロトコル制約を守りながら対象となる不具合を発見しやすくなる．

また今回のファジングにおいていくつかの課題が見て取れた．まず Peach にクラッシュ検知システムが用意されていない点だ．事例 1 ではソケットエラー，事例 2 では publisher 上に ping による死活監視を設定してクラッシュを判定した．ファジングにおいて死活監視はほとんどテスト上で行う．したがって Peach 上に標準の死活監視システムを用意しておくべきである．

次にファジング設計に対してプロトコルの知識やファジングの知識を要する点である．ファジングの採用意図としては労力の削減がある．したがって IoT で採用するにはファジング設計をできるだけ容易にする必要がある．

事例 1 では数回のファジングでエラーが検出され，ファジングの有効性が見て取れた．一方でクラッシュの原因がわからないという問題が発生した．これは今回のファジングにおいて数回のファジングで不具合を検出したことや Peach においてファジングを繰り返す機能がないことにより，異常系，正常系のパターンがともに少なく原因究明が困難であったからである．したがって Peach 上にクラッシュ検知後に繰り返す機能を追加する必要がある．また正常系や異常系のファズから原因究明を可視化する構造が必要である．

事例 2 では，Peach におけるファジングのいくつかの課題が明らかとなった．まず通信ごとに異なる初期値を設定できない点である．Peach には Pits ファイル上に通信を変化させる関数を呼び出す要素を持つ．これはテストデータ内の他の領域に合わせて値を変化させるものである．しかし今回のテストでは，テスト対象の入力に合わせて値を変化させる必要があった．したがってテスト対象からの入力を取得できる関数を用意し，Pits ファイルから呼び出せるようにすべきである．次に検出したファズから原因究明が容易にできない点である．事例 2 ではパターン 1 では 200 個のファズ，パターン 2 では 4 個のファズを発見した．一方でそれらのファズには，不具合の原因以外のデータも付いており，どこが不具合の直接的原因なのか把握する事ができない．したがって最小限のファズを特定できる機構が必要となる．この問題の解決策として考えられるのは文法規則ごとにファズを解析する方法である．図 5 はパターン 1 における各ファズでのコントロールフラグの出現回数を合計したものである．今回のような固定長ヘッダのファジングにおいて，各領域で示される値の数は有限である．またファズにはクラッシュの原因となった値が含まれているため，クラッシュしたファズを各文法規則に従って集計すれば不具合の原因となった値が多く出現することが考えられる．実際に，今回原因となったコントロールフラグ部を集計すると，RST フラグの出現回数が多いことがわかる．次に考えられる問題は，十分なファジング回数がわからない点である．特に今回利用した Peach はブラックボックスファザーであり，網羅的にファジングすることが不可能である．したがって十分な回数を見積もる方法を考える必要がある．

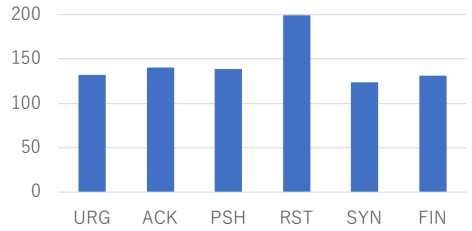

図 5 コントロールフラグの頻度

5.2 課題の分類

前節でファジングをテスト対象に適応させるにあたっていくつかの問題点を挙げた．本節で問題点を分類する．これらの問題点は大きく分けて 4 つに分けられる．それは 1 つはファジング適用時に手間がかかること，2 つ目は不具合の原因究明が困難であること，3 つ目は発見できない不具合があること，4 つ目はテストデータのデータ構造によっては不具合を発見できないことである．1 つ目のファジング適用時の手間は，ファジング作成時に通信プロトコルの知識や Publisher 作成に Peach の仕組みを理解する必要があり，ファジングの利用が労力の削減につながるか疑問が残るためである．2 つ目の不具合の原因究明が困難であることは，これらは検出したファズの少なさやファズから不具合の直接原因を明らかにすることが困難であること

に起因する．特に Peach では，不具合の推測は完全にユーザに委ねられており，ファジングを汎用的に利用するためには，不具合の推測を助けるようなシステムは不可欠だと考えられる．3 つ目は発見できない不具合がある点である．これは事例 2 で発生したことで，事例 2 の不具合はタイミングに起因する不具合であり，ファジングの設計によっては不具合を発見することができなくなる．4 つ目はテストデータのデータ構造によっては不具合を発見できないことである．事例 2 の調査結果からわかるように与えるデータ構造によってクラッシュの割合は大きく変わる．またテストデータに通信ごとに異なる値となるデータを入れると Peach では通信ごとに異なる初期値を設定できないため，ほとんどのデータが有効でないものとなり，クラッシュを発見する効率が低下する．

6. 終わりに

本調査では，IoT システムにおけるセキュリティ問題を解決すべく，IoT システムに対するファジングツール Peach の有効性，そして IoT システムに Peach を適応するにあたっての課題を調査した．具体的には Peach を用いて，不具合を持つオープンソース上の 2 つの実際の IoT システムに対してファジングを行い，その有効性を調査した．その結果，2 つの事例ともファジングで不具合を発見でき，Peach における IoT システムに対するファジングの一定の有効性を確認できた．一方でファジングをテスト対象に適応させるにあたっていくつかの問題点が発見された．それは大きく分けて 4 つに分けられ，1 つ目はファジング適用時に手間がかかること，2 つ目は不具合の原因究明が困難であること，3 つ目は発見できない不具合があること，4 つ目はモデルデータのデータ構造によっては不具合を発見できないことである．これらの課題はテスト担当者がテスト対象に合わせて工夫して解決することが必要である．

今後の課題としてはこの問題点を解決するファジングツールの開発が必要である．これらの問題点を元とした開発の指針は以下の 3 点が考えられる．

(1) ファジングツールを利用しやすくする．

(2) クラッシュの原因究明を助ける．

(3) 発見できない不具合を検出可能にする．

(1) については，特に Pits ファイルにおけるモデルデータのデータ構造の設定を助けるシステムが必要である．モデルデータの設計はテスト効率に大きく関わるが，設計にはテスト対象の通信プロトコルと Peach，両方の深い理解が求められる．したがってモデルデータの設計難易度を下げるツールの開発は Peach の利用難易度を下げることに大きくつながることが考えられる．また (2) においては，Peach 上ではファズから不具合の原因を明らかにするのは完全にユーザに委ねられているためである．したがって不具合の原因究明を助けるようなシステムが必要であることが考えられる．(3) は，本研究においてはタイミングに起因する不具合を検出できるように改良することが求められる．しかし本研究では Peach で検出できる不具合とできない不具合を体系的に分類できていない．したがってまず体系的な不具合の分類よりどの不具合に Peach が有効で，どの不具合に Peach が有効でないか調査することが求められる．その後発見できない不具合の中で発生頻度が多い不具合から修正していくべきである．

謝辞 本研究を実施するにあたり，名古屋大学の宮木龍氏，東京都市大学の藤原賢二講師には，様々な面で助言をいただきました．深く感謝いたします．本研究は，JST さきがけ JPMJPR21PA の助成を受けました．

参考文献

[1] Anne H Ngu, Mario Gutierrez, Vangelis Metsis, Surya Nepal, and Quan Z Sheng. IoT middleware: A survey on issues and enabling technologies. *IEEE Internet of Things Journal*, Vol. 4, No. 1, pp. 1–20, 2016.

[2] Statista. Iot: Number of connected devices worldwide 2012–2025. 2012.

[3] Constantinos Kolias, Georgios Kambourakis, Angelos Stavrou, and Jeffrey Voas. Ddos in the IoT: Mirai and other botnets. *Computer*, Vol. 50, No. 7, pp. 80–84, 2017.

[4] Maialen Eceiza, Jose Luis Flores, and Mikel Iturbe. Fuzzing the internet of things: A review on the techniques and challenges for efficient vulnerability discovery in embedded systems. *IEEE Internet of Things Journal*, 2021.

[5] Xiaotao Feng, Ruoxi Sun, Xiaogang Zhu, Minhui Xue, Sheng Wen, Dongxi Liu, Surya Nepal, and Yang Xiang. Snipuzz: Black-box fuzzing of iot firmware via message snippet inference. CCS '21, p. 337–350, New York, NY, USA, 2021. Association for Computing Machinery.

[6] Poonam Pingale, Kalpana Amrutkar, and Suhas Kulkarni. Design aspects for upgrading firmware of a resource constrained device in the field. In *2016 IEEE International Conference on Recent Trends in Electronics, Information & Communication Technology (RTEICT)*, pp. 903–907. IEEE, 2016.

[7] V. M. Manes, H. Han, C. Han, S. Cha, M. Egele, E. J. Schwartz, and M. Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, Vol. 47, No. 11, pp. 2312–2331, nov 2021.

[8] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. IoTFuzzer: Discovering memory corruptions in iot through app-based fuzzing. In *NDSS*, 2018.

[9] Amir Makhshari and Ali Mesbah. IoT bugs and development challenges. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 460–472. IEEE, 2021.