Vol.2022-SE-210 No.14
2022/3/11

コードクローン検出に基づく IoTを対象とした自動パッチ生成

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

概要：IoT は軽量のプラットフォームやネットワークプロトコルがよく使用され，インターネットを介してサービスを提供するため，セキュアな設計が求められる．ソフトウェアシステムを対象とした研究において，68%の欠陥は，複数箇所を同一または類似の変更により修正をしたという報告がある．このような修正は，1 箇所の変更を行い，その変更を複製し，編集することで実現される．そのため，修正を自動化することができる可能性があると考えられる．本研究では，セキュアな設計が必要な IoT における，同一または類似の変更により修正された欠陥を対象として，自動パッチ生成ができるか調査を行った．その結果，88%の欠陥に対して自動的にパッチを生成することができた．

Automatic Patch Generation for IoT Based on Code Clone Detection

Kentaro Ohno[1,a)]　Norihiro Yoshida[1,b)]　Wenqing Zhu[1,c)]　Hiroaki Takada[1,d)]

1. はじめに

Internet of Things (IoT) は，標準の通信プロトコルを使用して，デバイスをインターネットに相互接続することで，複雑なネットワーク構築を行う [1], [2]．相互接続されたデバイスは，デジタル世界での物理的または仮想的な表現，検知/作動機能，プログラム可能な特徴を備えており，一意に識別が可能である．IoT 接続デバイスの数は 5 年ごとに倍増しており [3]，2025 年までに世界中で 750 億を超える IoT 接続デバイスが存在すると推定されている [2], [4]．

IoT に接続されたエッジデバイスの計算リソースは限られており，軽量のプラットフォームやネットワークプロトコルがよく使用される．軽量のプラットフォームやネットワークプロトコルは攻撃に対する耐性が低く，開発者はセキュアなソフトウェア開発が求められる．また，IoT はインターネットを介してサービスを提供するため，従来のシステムと比べて複雑性が増しており [5]，システムに発生する脆弱性などの欠陥を含む可能性が高いといわれている [6]．そのため，IoT におけるソフトウェア上に含まれる欠陥を容易かつ早急に検出・修正することが求められる．

現在までに，コードクローン検出を用いて欠陥検出を行う開発が進められている [7], [8], [9], [10]．たとえば，Kim らはスケーラビリティを持ち，正確なコードクローン検出へのアプローチである VUDDY[8] を提案した．これは，関数に堅牢な解析と新しいフィンガープリントメカニズムを採用している．Gao らは，IoT 関連の欠陥検出として IoTSeeker[10] を提案した．これは，バイナリコードからラベル付きのセマンティックフローグラフを作成し，セマンティック特徴をベクトルとしたニューラルネットワークモデルを構築することで実現する．加えて，ソフトウェアプログラム解析におけるコミュニティでは，自動プログラム修正として，対象となる欠陥が含まれるプログラムに対して，制約条件を満たし，テストが成功するようにプログラムを変異させるアプローチで研究が進められてきた．これらのアプローチは一定の有効性を示す一方で，修正可能な欠陥の数，パラメータの調整，複数行の修正，テストケースの数における課題を持つ．本論文では，自動プログラム修正の従来とは異なるアプローチとして，コードクローン

[1] 名古屋大学
Furo-cho, Nagoya, Aichi 464-8601, Japan
[a)] k_ohno@ertl.jp
[b)] yoshida@ertl.jp
[c)] zhuwqing1995@ertl.jp
[d)] hiro@ertl.jp

検出に基づく自動パッチ生成を提案する．

この提案手法は，コードクローン検出による欠陥箇所の特定とプログラム修正となるパッチの生成を自動的に行う．本研究では，容易かつ早急な対応が求められる IoT の欠陥に適用することで提案手法の有効性の評価を行った．

本研究の貢献は，以下の 2 点である．

- 様々なプログラム構造に一般化した自動パッチ生成手法を提案した．
- IoT の欠陥事例を対象とした適用調査を行い，88%の事例で自動パッチ生成を実現し，高い有効性を示した．

2 章では本手法で使用するツールと提案手法のアプローチの可能性を示した関連研究の紹介を，3 章では提案手法のアプローチを，4 章では IoT の欠陥を対象とした提案手法の評価を，5 章では本研究のまとめを記す．

2. 関連研究

2 章では，本手法で使用するコードクローン検出ツール CCFinderSW[11][12] と，編集スクリプト生成ツール GumTree[13] を紹介する．CCFinderSW は欠陥検出のため，GumTree はコードクローンの追加や削除となっているトークンを把握するために使用する．また，本手法の可能性を示した研究として，Madeiral らの研究 [14] を紹介する．

2.1 コードクローン検出ツール CCFinderSW

ソフトウェアプロジェクトにおける欠陥の伝播などのコードクローンの問題を解決するために，様々なコードクローン検出ツールが開発されている．ソースコードの正規化と類似度の計算によって，ツールの検出能力は異なる．一般的な特徴として，検出可能なコードクローンのタイプと検出単位がある．さらに，多くのツールは，コード片の類似度をパラメータ化することにより，類似度を設定できる．類似度のしきい値が低いほど，より欠陥を含むコードクローンが検出される可能性があるが，より多くのコード片が誤ってコードクローンとして報告される．本手法で使用する CCFinderSW[11][12] は，コードクローン検出ツールの一つで，パーサジェネレータ生成システムの 1 つである ANTLR の構文定義記述を入力として指定することで，さまざまなプログラミング言語でコードクローンを検出することができる．CCFinderSW は，字句解析，変数処理，およびコードクローン検出/出力シェーピングの 3 つのステップでコードクローン検出を実行する．

2.2 編集スクリプト生成ツール GumTree

GumTree[13] は，端的に元の開発者の意図に近い編集スクリプトを計算することを目的とした，移動，追加，削除を含む抽象構文ツリーの粒度で編集スクリプトを計算するアルゴリズムを導入することにより，ツリー間の差を計算するツールである．行の追加と行の削除のアクションのみを使用したテキスト行の粒度で編集スクリプトを計算するのではなく，移動，追加，削除を含む抽象構文ツリーの粒度で計算することで，編集スクリプトから構文の変更を推測することが容易となる．これは，コード片の移動がソースコードを編集するときに頻繁に実行されるアクションである点を考慮に入れたアルゴリズムである．本手法では，コードクローンに対して，GumTree を適用し，テキストベースの出力ファイルから，追加，削除となっているトークンを抽出し，新パッチ生成に適用する．

2.3 Madeiral らの研究

Madeiral らの研究 [14] は，欠陥を修正するために開発者がコードクローンをどの程度適用するかを明らかにするために，96 のプロジェクトの 10,290 のパッチに含まれる欠陥修正の変更をまとめたデータセットから 3,049 個の複数の隣接していない場所での変更で構成されるパッチを手動で分析し，パッチにおける修正のコードクローングループを，以下に定義する 5 種類に分類した．

- P-A: 図 1(a) に示すように，変更は同一であり，コンテキスト（変更部分の前後のコード片）も同一なパッチである．
- P-B: 図 1(b) に示すように，変更は同一であり，コンテキストは構造的に類似しているパッチで，識別子・リテラル・型・レイアウト・コメント・追加または削除されたコード片が含まれている可能性があるパッチである．
- P-C: 図 1(c) に示すように，変更は構造的に類似しているパッチで，識別子・リテラル・型・レイアウト・コメント・追加または削除されたコード片が含まれている可能性があるが，コンテキストは同一なパッチである．
- P-D: 図 1(d) に示すように，変更とコンテキストは構造的に類似しており，識別子・リテラル・型・レイアウト・コメント・追加または削除されたコード片が含まれている可能性があるパッチである．
- P-E: 図 1(e) に示すように，変更は同一であるが，コンテキストは構造的に類似していないパッチである．

修正のコードクローンはパッチに頻繁に存在し，分析された複数の隣接していない場所での変更で構成されるパッチの 68%に存在し，それらの 70%は修正のコードクローンのみの変更で構成されている．さらに，修正のコードクローンのみの変更で構成されたパッチの 89%には，同一の変更を加えた修正のコードクローン（P-A または P-B）が含まれていることを明らかにした．一方で，Madeiral らの研究は，複数箇所に対して類似した修正を行うケースは自動化できる可能性が高いことを示唆しているが，自動化のための手法の提案には至っていない．この理由として，

```
- joiner.add(arg.toString());
+ joiner.add(String.valueOf(arg));
...
- joiner.add(arg.toString());
+ joiner.add(String.valueOf(arg));
```

(a) P-A

```
- password = new TextField<String>();
+ password = new KapuaTextField<String>();
...
- confirmPassword = new TextField<String>();
+ confirmPassword = new KapuaTextField<String>();
```

(b) P-B

```
- ((AccountImpl) account).setScopeId(scopeId);
+ account.setScopeId(scopeId);
 ...
- ((CredentialImpl) credential).setScopeId(scopeId);
+ credential.setScopeId(scopeId);
```

(c) P-C

```
- callsPerKey *= numKey1 / (double) numRecords1;
+ callsPerKey *= (double)numRecords1 / numKey1;
...
- callsPerKey *= numKey2 / (double) numRecords2;
+ callsPerKey *= (double) numRecords2 / numKey2;
```

(d) P-D

```
- response.write (data + "<||>");
+ response.write (data + END);
...
- return message + "<||>";
+ return message + END;
```

(e) P-E

図 1: パッチのタイプ別分類

Madeiral らは，様々なプログラム構造に一般化できる自動プログラム修正において考えられる課題と解決策が未検討である点を挙げている．

様々なプログラム構造に一般化した自動プログラム修正の 1 つの解決策として，本手法を提案する．

3. 提案手法

3 章では，本手法のアプローチを記す．本手法は，ユーザがソースコード上に含まれる 1 つの欠陥を修正した後，コードクローン検出ツールを使用することで，対象としているプログラムに含まれる修正した欠陥に類似したコードクローンを検出し，検出された欠陥のコードクローンに対して自動的にパッチを生成する手法を提案する．これは，以上のように，ユーザが 1 つの欠陥を修正したことを前提としている．そのため，1 つの欠陥を修正した前後の差分を抜き出したパッチファイルと，修正前のコード片を含む対象とするプログラムが格納されているディレクトリ名を入力として与える．Listing 1 にパッチファイルの例を示す．

```
--- a/.../Accounts.java
+++ b/.../Accounts.java
@@ -236,1 +235,1 @@ public Account update(
- ((AccountImpl) account).setScopeId(scopeId);
+ account.setScopeId(scopeId);
```

Listing 1: パッチファイルの例

```
#begin{set}
0.2 134,3,711 172,71,844 91
0.2 210,3,950 248,71,1083 91
0.3 135,3,716 173,71,849 91
0.3 211,3,955 249,71,1088 91
0.6 159,3,832 197,71,965 91
0.6 235,3,1071 273,71,1204 91
0.7 134,3,729 172,71,862 91
0.7 207,3,968 245,71,1101 91
0.9 130,3,729 168,71,862 91
0.9 206,3,968 244,71,1101 91
0.11 130,3,713 168,71,846 91
0.11 206,3,952 244,71,1085 91
#end{set}
```

Listing 2: CCFinderSW における出力結果例

本手法は，2 段階の処理に分かれている．第 1 段階では，コードクローン検出ツール CCFinderSW[11][12] を用いて，ユーザが修正した欠陥とのコードクローンを検出する．CCFinderSW を採用した理由は，事前調査で欠陥検出において有効性を示した点，テキストベースのファイルを出力とするため 2 次利用が容易である点からである．Listing 2 に CCFinderSW における検出結果例を示す．**図 2** に示した欠陥検出の流れのように，以下の順序に従って，検出された欠陥のコードクローンが含まれるソースファイル名，コード片を得る．

(1) パッチファイルから，「ソースファイル名」，「修正するソースコード行数」を抽出する．
(2) CCFinderSW の出力ファイルにおいて，「ソースファイル名」と「ファイル番号」を抽出し，紐づける．
(3) パッチファイルで抽出した「ソースファイル名」と紐づけた「ファイル番号」と「修正するソースコード行数」を含む CCFinderSW の出力ファイルにおける箇所を抽出する．
(4) 抽出した CCFinderSW の出力ファイルにおける箇所

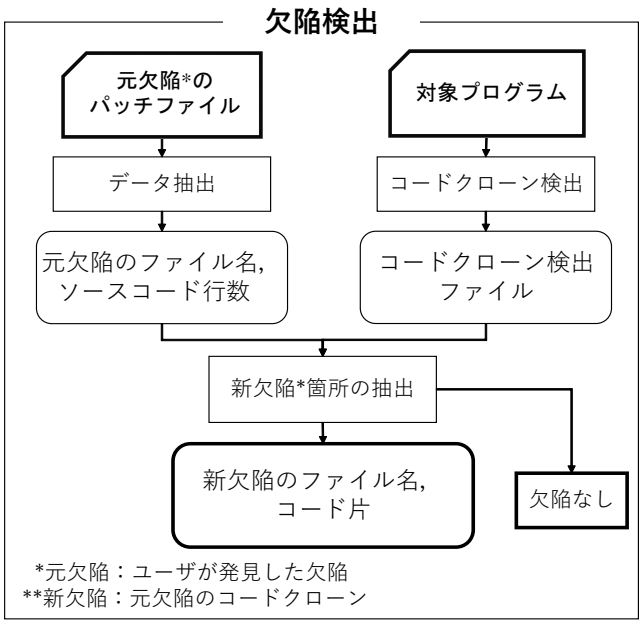


図 2: 欠陥検出の流れ

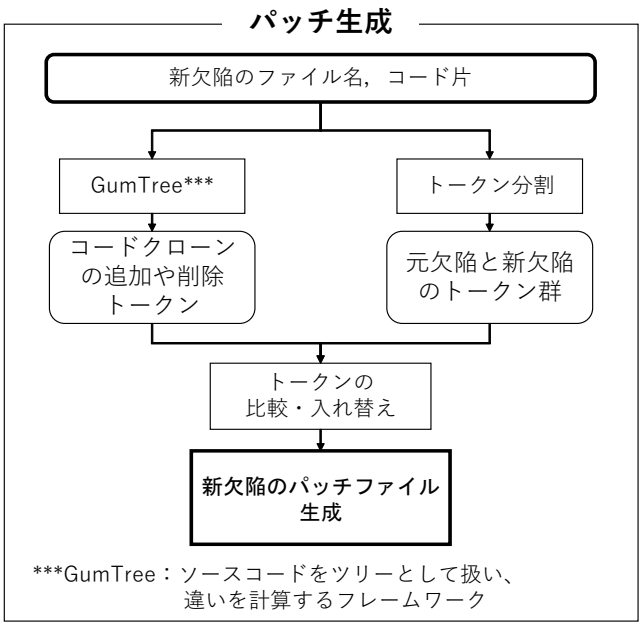


図 3: パッチ生成の流れ

```
===
insert-node
---
SimpleName: detachTimeOut [103,116]
to
METHOD_INVOCATION_ARGUMENTS [88,115]
at 2
```

Listing 3: GumTree における挿入トークンの出力結果例

の前後に記述された「欠陥のコードクローン」を特定する．

(5) 検出された「欠陥のコードクローン」が含まれる「ソースファイル名」,「ソースコードの行数」を得る．

(6)「ソースファイル名」と「ソースコードの行数」から，検出された「欠陥コードクローンのコード片」を抽出する．

欠陥のコードクローンが検出されなかった場合，第 2 段階に進まず，第 1 段階で終了する．第 2 段階では，元の欠陥と欠陥のコードクローンをトークン単位に分割・比較し，異なるトークンを入れ替えることで，元パッチを編集し，新パッチを生成する．編集スクリプト生成ツール GumTree[13] を用いて抽出した挿入・削除のトークンがある場合は比較せず，新パッチに反映させる．Listing 3 に GumTree における挿入トークンの出力結果例を示す．**図 3** に示したパッチ生成の流れのように，以下の手順により，検出された欠陥のコードクローンに対して自動的にパッチを生成する．

(1) パッチファイルから,「修正前のコード片」,「修正後のコード片」を抽出する．追加のみの修正の場合は,「修正後のコード片」のみ抽出する．

(2) 第 1 段階で抽出した「欠陥コードクローンのコード片」と，パッチファイルから抽出した「修正前のコード片」を GumTree にかけることで,「追加・削除されているトークン」を抽出する．追加のみの修正の場合は，第 1 段階で抽出した欠陥コードクローンを含むスコープ単位で行う．

(3) 第 1 段階で抽出した「欠陥コードクローンのコード片」とパッチファイルから抽出した「修正前のコード片」をトークン単位に分割する．

(4) 分割したトークン同士で順番に比較していく．異なるトークンの場合，修正後のコード片のトークンを入れ替える．このとき,「追加・削除のトークン」がある場合は比較せず,「修正後のコード片」に「追加・削除のトークン」を反映させる．

(5)「欠陥のコードクローン」の特徴を反映させたパッチファイルを生成し，ユーザに提供する．

図 4 に追加・削除を含む場合における修正前の欠陥と欠陥コードクローンの比較例を，**図 5** にトークンの入れ替えが発生する場合における修正前の欠陥と欠陥コードクローンの比較例を示す．

4. ケーススタディ

4 章では，提案手法の有効性を確認するために，IoT の欠陥事例を対象とした自動パッチ生成における評価実験について記す．

4.1 IoT の欠陥事例の収集

IoT の欠陥事例の収集にあたり，Makhshari らのデータセット [2] を利用した．Makhshari らのデータセットは，IoT に関わるプロジェクトの欠陥レポート（イシュー）を

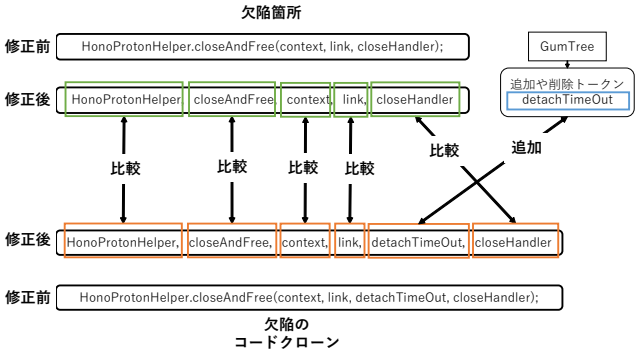


図 4: 追加・削除を含む場合における比較例

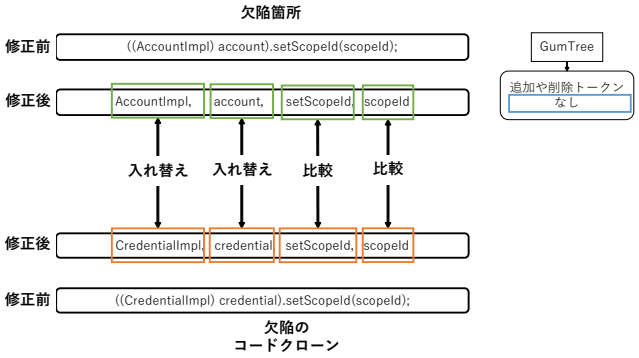
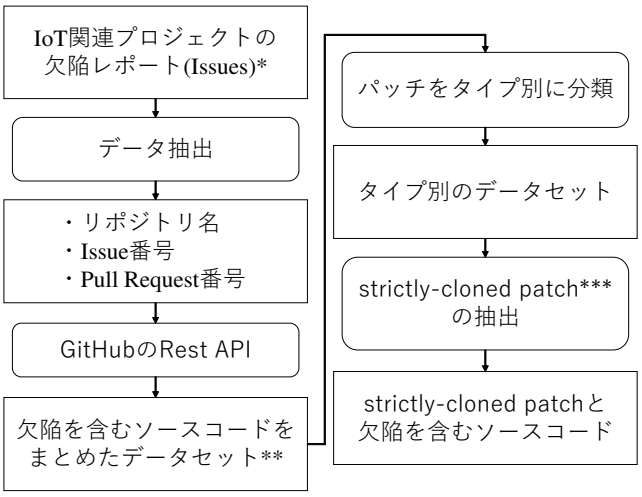


図 5: トークンの入れ替えが発生する場合における比較例

収集したデータセットである．以下のような流れで，関連するプルリクエストにおける変更した差異を記述したパッチファイルと修正前のソースコードを収集し[*1]，その中から，修正箇所がコードクローンとなっている IoT の欠陥事例を目視で収集した．また，IoT の欠陥事例を [14] で定義されているようにパッチのタイプ P-A, P-B, P-C, P-D に分類した．さらに，修正箇所がコードクローンとなっている IoT の欠陥事例の中でも，strictly-cloned patches[14]（すべての修正がコードクローンとなっているパッチ）をタイプ P-A, P-B, P-C, P-D 毎に抽出した．

(1) データセット [15] からリポジトリのオーナー名，リポジトリ名，イシュー番号を取得する．
(2) 次に，REST API[16] を使用して，オーナー名，リポジトリ名，イシュー番号からプルリクエスト番号を検索する．これは，イシュー番号から直接パッチファイルとパッチファイルにより変更されたファイル群を取得することはできないためである．また，イシューとプルリクエストの関係は一様ではないので，すべてのプルリクエスト番号を取得することはできない．
(3) 取得したオーナー名，リポジトリ名，プルリクエスト番号から GitHub の REST API を使用して，パッチファイルとパッチにより変更されたファイル群を取得する．
(4) パッチファイルとパッチにより変更されたファイル群を入力として，修正される前のファイル群をコマンドにより取得する．
(5) パッチファイルを目視で確認し，修正箇所がコードクローンとなっている IoT の欠陥事例を手動で収集する．
(6) 修正箇所がコードクローンとなっている IoT の欠陥事例をタイプ P-A, P-B, P-C, P-D に分類する．また，それぞれのタイプから strictly-cloned patch を抽出する．

[*1] https://zenodo.org/record/5090430#.YgZcit_P02w



* IoT関連プロジェクトの欠陥レポート(Issues)
：https://github.com/IoTSEstudy/IoTbugschallenges
** 欠陥を含むソースコードをまとめたデータセット
：https://zenodo.org/record/5090430.YOqPOj7Q2w
*** strictly-cloned patch
：コードクローンのみで構成された欠陥修正で構成されたパッチ

図 6: IoT に関する欠陥事例の収集フロー

本研究の評価では，修正箇所がコードクローンとなっている IoT の欠陥事例の中で，すべての修正がコードクローンとなっている 41 事例（strictly-cloned patches を持つ欠陥事例）を対象とした．IoT の欠陥事例の収集の流れを**図 6** に示す．

4.2 評価指標

本研究では，コードクローンとなっている修正箇所の 1 つを元パッチとして用いることで，それ以外のコードクローンとなっている修正箇所を新パッチとして生成することができれば，自動パッチ生成成功と定義する．また，以下のように適合率，再現率を定義する．

- 適合率＝パッチ生成成功数/パッチ生成数
- 再現率＝パッチ生成成功数/欠陥検出数

適合率は，パッチが生成された中でどれだけ正しかったかを示し，コードクローン検出による欠陥の誤検出とパッチ生成失敗に影響している．再現率は，パッチが生成された中でどれだけ取りこぼしがないかを示し，パッチ生成成功率を表す．欠陥検出率は，コードクローン検出によって，どれだけ取りこぼしなく欠陥検出を行えたかを示し，コードクローン検出における欠陥検出能力を表す．また，パッ

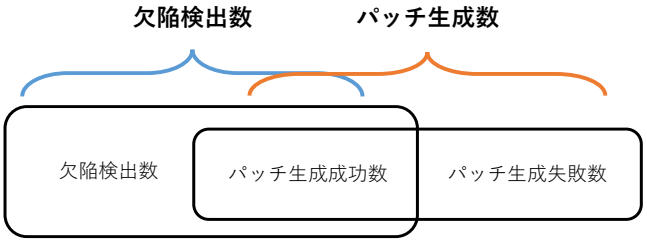


図 7: パッチ生成成功数，パッチ生成数，欠陥検出数の関係

チ生成成功数，パッチ生成数，欠陥検出数の関係を**図 7** に示す．

4.3 結果

ケーススタディの結果を**表 1** に示す．本手法は IoT の 41 の欠陥事例に対して，63%の欠陥検出に成功した．また，欠陥検出された IoT の欠陥事例における，88%の事例に対して，パッチ生成に成功した．これらの結果は，本手法である自動パッチ生成において高い有効性を示した．P-A, P-B, P-C は再現率が 100%に対して，P-D は再現率が 57%であった．これらの原因を 4.4.1 で考察する．加えて，自動パッチ生成の適合率における考察を 4.4.2，コードクローン検出による欠陥検出における考察を 4.4.3 で記す．

4.4 考察

4.4.1 自動パッチ生成の再現率

本調査では，コードクローン検出された IoT の欠陥事例に対して，再現率は 88%となり，本手法の有効性を示した．一方で，パッチを生成できなかった事例がある．この原因として，関数呼び出しの順序が入れ替わっている場合が挙げられる．この事例を**図 8** に示す．このような事例において，本手法では対応できていない．このような事例の対応策の 1 つとして，ユーザが関数呼び出しの順序を入れ替える対策も考えられるが，関数呼び出しの順序を入れ替えられない場合も存在する．関数呼び出しの順序を一時的に入れ替えて，パッチを生成した後に，元の関数呼び出しの順序に入れ替えるなど，アプローチ内の改善が求められる．また，本調査では，41 の IoT の欠陥事例に対して，自動パッチ生成を試みた．事例を増やすことにより，多数の事例での本手法の有効性を示すことが求められる．

4.4.2 自動パッチ生成の適合率

本調査では，コードクローンとして検出された IoT の欠陥事例に対して，60%の適合率を示した．これは，パッチ生成において 40%が実際にパッチとして適用されないことを表し，手動でパッチを適用するユーザに対する負担が増えてしまうという課題がある．この課題の改善策の 1 つとして，自動生成したパッチを対象プログラムに適用して，新たな不具合が起きないかテストを行い，テストを通過したパッチのみユーザに提供する方法が挙げられる．このとき，実際にパッチを適用するかどうかはユーザが判断するため，適用しない場合パッチ適用前に戻す必要がある．このようなアプローチ内の改善により，パッチ適用時のユーザの負担を抑えることができると考えられる．

```
@@ -122,6 +123,7 @@ public void handleEvent(BaseEvent be) {
 startsOnTime.setStyleAttribute("padding", "0px 0px 0px 17px");
  startsOnTime.setEmptyText(JOB_MSGS.dialogAddScheduleTimePlaceholder());
  startsOnTime.setToolTip(JOB_MSGS.dialogAddScheduleStartsOnTimeTooltip());
+ startsOnTime.setTriggerAction(TriggerAction.ALL);
  startsOnPanel.add(startsOnTime);
  mainPanel.add(startsOnPanel);

@@ -149,6 +151,7 @@ public void handleEvent(BaseEvent be) {
  endsOnTime.setEmptyText(JOB_MSGS.dialogAddScheduleTimePlaceholder());
  endsOnTime.setToolTip(JOB_MSGS.dialogAddScheduleEndsOnTimeTooltip());
 endsOnTime.setStyleAttribute("padding", "0px 0px 0px 17px");
+ endsOnTime.setTriggerAction(TriggerAction.ALL);
  endsOnPanel.add(endsOnTime);
  mainPanel.add(endsOnPanel);
```

図 8: パッチ生成失敗の事例

4.4.3 コードクローン検出による欠陥検出

本手法は，コードクローン検出による欠陥検出を第一段階としている．欠陥検出の割合を上げることは，ソフトウェアセキュリティにおいて重要である．本調査では，41 の IoT の欠陥事例に対して 63%の欠陥検出率を示した．欠陥検出率に関する要因の 1 つ目として，類似度などの設定値がある．類似度の値以上が類似しているコード片をコードクローンとして判断する．本手法で使用した CCFinderSW[11][12] は最小一致トークン数を類似度の指標として用いている．標準設定では，最小一致トークン数は 50 トークンに設定されており，任意の範囲の中で 50 トークン以上類似したトークンが存在すればコードクローンとして検出される．この最小一致トークン数（類似度）を低くすることで，欠陥検出率が高くなる可能性がある．一方で，より抽象的な検出となってしまい，欠陥の誤検出率も高くなる可能性がある．これは，ユーザの作業が増えてしまうといった問題点を含み，トレードオフの関係にあるため，IoT の欠陥に対する最適な設定が求められる．欠陥検出率に関する要因の 2 つ目として，コードクローンの検出能力がある．検出能力には，コメント，レイアウト以外全て一致するコード片（タイプ 1）の検出，タイプ 1 に加えて，識別子，識別子の型の変更したコード片（タイプ 2）の検出，タイプ 2 に加えて，ステートメントの追加・削除をしたコード片（タイプ 3）の検出などがある．本手法で使用した CCFinderSW はタイプ 2 の検出を行うコードクローン検出ツールである．検出能力においても，タイプ 3 の検出を行うコードクローン検出ツールを用いて欠陥検出をすることで欠陥検出率が高くなる可能性があるが，欠

表 1: 自動パッチ生成の評価結果

パッチタイプ	欠陥事例数	欠陥検出数	パッチ生成成功数	欠陥検出率	適合率	再現率
P-A	15	8	8	53%	64%	100%
P-B	14	8	8	57%	85%	100%
P-C	3	3	3	100%	45%	100%
P-D	9	7	4	78%	80%	57%
合計	41	26	23	63%	60%	88%

陥の誤検出率も高くなる可能性があり，トレードオフの関係にあるため，注意が必要である．

5. まとめ

本研究は，ユーザが 1 つの欠陥を修正した後，その欠陥と類似している欠陥を検出し，自動的にパッチを生成する手法を提案した．IoT の欠陥事例を対象とした自動パッチ生成における評価実験の結果は，適合率は 60%，再現率は 88%であり，本手法の有効性を確認した．一方で，関数呼び出しの順序の入れ替わりなどの理由から自動パッチ生成ができなかったことが分かり，手法の改善が必要である．また，本研究ではパッチ生成を行ったが，パッチ適用時における障害の調査をするパッチ適用テストの必要性や，コードクローン検出ツールの設定変更の簡易性，UI の充実などユーザの使いやすさに関連した検討が必要である．

謝辞 本研究は，JST さきがけ JPMJPR21PA および JSPS 科研費 JP20K11745 の助成を受けた．

参考文献

[1] Roberto Minerva, Abyi Biru, and Domenico Rotondi. Towards a definition of the Internet of Things (IoT). *IEEE Internet Initiative*, 1:1–86, 2015.

[2] Amir Makhshari and Ali Mesbah. IoT bugs and development challenges. In *Proc. of ICSE 2021*, pages 460–472, 2021.

[3] Katie Costello. Predicts 2021: Cloud and edge infrastructure cloud infrastructure edge. `https://www.gartner.com/smarterwithgartner/gartner-predicts-the-future-of-cloud-and-edge-infrastructure`, 2021. (Accessed on 02/14/2022).

[4] Statista. Internet of Things (IoT) connected devices installed base worldwide from 2015 to 2025 (in billions).

[5] Mark Hung. Leading the IoT, gartner insights on how to lead in a connected world. *Gartner Research*, 1:1–5, 2017.

[6] Xingbin Jiang, Michele Lora, and Sudipta Chattopadhyay. An experimental analysis of security vulnerabilities in industrial IoT devices. *ACM Transactions on Internet Technology (TOIT)*, 20(2):1–24, 2020.

[7] Nam H Pham, Tung Thanh Nguyen, Hoan Anh Nguyen, Xinying Wang, Anh Tuan Nguyen, and Tien N Nguyen. Detecting recurring and similar software vulnerabilities. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 2, pages 227–230. IEEE, 2010.

[8] Seulbae Kim and Heejo Lee. Software systems at risk: An empirical study of cloned vulnerabilities in practice. *Comput. Secur.*, 77:720–736, 2018.

[9] Norihiro Yoshida, Takeshi Hattori, and Katsuro Inoue. Finding Similar Defects Using Synonymous Identifier Retrieval. In *Proceedings of the 4th International Workshop on Software Clones*, IWSC '10, page 49–56, New York, NY, USA, 2010. Association for Computing Machinery.

[10] Jian Gao, Xin Yang, Yu Jiang, Houbing Song, Kim-Kwang Raymond Choo, and Jiaguang Sun. Semantic learning based cross-platform binary vulnerability search for iot devices. *IEEE Transactions on Industrial Informatics*, 17(2):971–979, 2019.

[11] Yuichi Semura, Norihiro Yoshida, Eunjong Choi, and Katsuro Inoue. CCFinderSW: Clone Detection Tool with Flexible Multilingual Tokenization. In *Proc. of APSEC 2017*, pages 654–659, 2017.

[12] Yuichi Semura, Norihiro Yoshida, Eunjong Choi, and Katsuro Inoue. Multilingual Detection of Code Clones Using ANTLR Grammar Definitions. In *Proc. of APSEC 2018*, pages 673–677, 2018.

[13] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 313–324, 2014.

[14] Fernanda Madeiral and Thomas Durieux. A large-scale study on human-cloned changes for automated program repair. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 510–514. IEEE, 2021.

[15] Amir Makhshari and Ali Mesbah. IoTSEstudy/IoTbugschallenges: Replication Package of "IoT Bugs and Development Challenges" study. `https://github.com/IoTSEstudy/IoTbugschallenges`. (Accessed on 02/14/2022).

[16] GitHub. GitHub REST API - GitHub Docs. `https://docs.github.com/en/rest`. (Accessed on 02/14/2022).