Vol.2021-SPT-41 No.8
2021/3/1

F*言語によるMQTTパケットパーサの開発と安全性評価

喜多村 卓[1,a)]　上原哲太郎[2,b)]

概要：IoT 製品においては, 製品出荷後のバグ修正が困難なシステムなど, 開発時点でバグがあってはならないシステムが存在する. このようなシステムは, プログラム開発時点で堅牢なプログラムであることは重要である. プログラムの堅牢性を保証する開発手法として形式手法が提案されているが, 実際の開発現場では普及しているとは言えない. 本研究では, 形式手法普及の阻害要因を明らかにする目的で, プログラム検証用言語である F*言語を用いて堅牢なシステムであることが求められる MQTT パケットパーサを開発し, その過程で明らかになった課題を整理するとともに, 安全性評価を行った.

Development and Safety Evaluation of MQTT Packet Parser Using F* Language

1. はじめに

IoT 製品, 特に民生機器における課題の一つに, 製品出荷後にソフトウェアにバグが発見された場合の対応が挙げられる. パソコン等と異なり多くの場合ユーザインターフェースに乏しい IoT 機器においては, ソフトウェア更新の可否や更新タイミングを利用者に指定させることは困難であるため, バグ改修のためのソフトウェア更新が大幅に遅れたり, いつまでも更新されないような事態も多発する. バグがソフトウェアのセキュリティ上の脆弱性に繋がっている場合には, 当該機器への不正アクセスやマルウェア侵入の危険がある. そのため, そもそもセキュリティ脆弱性の発生する余地が少ない, すなわちセキュリティ面から見て堅牢なソフトウェアが IoT 機器には求められている. 堅牢なソフトウェアの開発には, 形式手法などいくつかの提案がすでに知られている. しかし図 1 によると, 製品出荷後にソフトウェアを更新することが難しい産業用制御システムの脆弱性報告件数は増加傾向にある. これはすなわち, 産業用制御システムのソフトウェアの開発現場においては堅牢なソフトウェアの開発手法が十分に普及しているとは言いがたいことを示しており, このようなシステムの IoT 化は現状ではセキュリティリスクが高いことを表している. そこで, 本研究では, IoT 機器などソフトウェア更新が困難な機器における形式手法を用いたソフトウェア開発にかかる課題を明らかにするため, 堅牢なシステムであることが求められる MQTT パケットパーサをプログラム検証用言語である F*言語を用いて開発し, 性能評価を行った. その過程で明らかになった課題を元に, 形式手法をこのようなシステムの開発に適用する際の困難性について評価した.

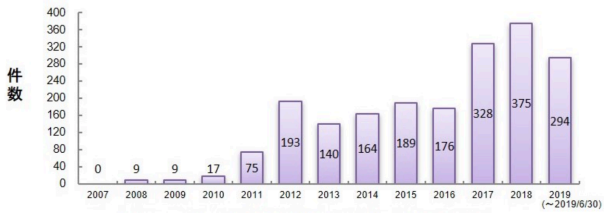

図 1　産業用制御システムの脆弱性対策情報件数 [1]

1.1 形式手法

形式手法とは, システム全体を抽象化してテストを行う手法の総称であり, 堅牢なシステム開発に役立つとされる. 形式手法は, システムの仕様を厳密に書くことを目的とした形式仕様記述, システムの状態遷移を検証することを目的としたモデル検査の大きく 2 つに分類できる. 本研究で

[1] 立命館大学大学院情報理工学研究科
Graduate School of Information Science and Engineering, Ritsumeikan University
[2] 立命館大学情報理工学部
College of Information Science and Engineering, Ritsumeikan University
[a)] kitamura@cysec.cs.ritsumei.ac.jp
[b)] t-uehara@fc.ritsumei.ac.jp

© 2021 Information Processing Society of Japan

はこのうち, 形式仕様記述に分類される定理証明と呼ばれる技術を活用するため, F*言語と呼ばれるプログラム検証言語を採用することにした.

1.2 F*言語

F*言語 [2] はプログラムの検証を目的として, Microsoft Research, MSR-INRIA, INRIA らによって開発中の関数型言語である. F*言語では, 各関数の引数と返り値がそれぞれ満たすべき制約条件を記述すると, プログラム全体が制約条件と矛盾が無いか検証することができる. 各関数の引数と返り値は一般にプログラムの詳細設計書にその機能仕様が書かれているため, 仕様に誤りがないと仮定すると, そのプログラム全体が仕様と矛盾しない, すなわちバグがないことが保証できる. F*言語でプログラムを開発する際は, 関数の制約条件を満たすようにコードを記述・修正するだけでよいため, 無駄な処理が入り込みにくい. F*言語によるプログラムからは, C 言語によるプログラムを自動生成可能であり, 高い実行性能が期待できる. そこで本研究では, 検証済みかつ実行性能の高いプログラムを開発可能な F*言語を, 堅牢なシステムであることが求められる MQTT パケットパーサの開発言語として採用することにした.

1.2.1 Project Everest

公開鍵認証基盤 PKI と TLS は, インターネットのセキュリティ基盤として広く利用されている技術である. Project Everest[3] は, インターネットで最も頻繁に利用されるプロトコル HTTP を TLS で暗号化した HTTPS にも脆弱な部分があるとして, HTTPS の構成要素となるソフトウェアモジュールを, F*言語を用いて検証済みのものへ置き換えることを目的として Microsoft Research, カーネギーメロン大学, INRIA, MSR-INRIA らが主導しているプロジェクトである. Project Everest では現在, 以下のようなソフトウェアモジュールを F*言語を用いて実装するプロジェクトが進行している.

miTLS[4]
: TLS1.3 の検証済みリファレンス実装

HACL*[5]
: 'Mozilla Firefox' や'WireGuard' などでの利用実績がある検証済み暗号ライブラリ

Vale[6]
: 暗号化コードに特化した検証済みアセンブリ言語を生成可能なソフトウェア

1.3 MQTT

MQTT は低帯域幅や高遅延であるなど, 信頼性の低いネットワークにも対応できるよう設計された IoT 機器向けの通信プロトコルである. 近年 IoT 機器の普及に伴い, サイバー攻撃の対象にされると考えられているため, MQTT プロトコルスタックの実装も堅牢であることが求められている. MQTT プロトコルスタックへのサイバー攻撃は, まずパケットパース部に向けて行われることが想定されるが, MQTT パケットへの細工による異常な入力は無数に考えられるのでパケットパース部がこれらの異常な入力を正しく排除できるようにプログラムを記述することは困難である. そこで, 本研究では F*言語を用いることで, 仕様を満たさない異常なパケットを正しく排除できる堅牢な MQTT パケットパーサを開発し, 性能評価を行った. 本研究で開発した MQTT パケットパーサは, OASIS(Organization for the Advancement of Structured Information Standards) により 2019 年 3 月 7 日に標準化された MQTT Version 5.0[7] に準拠したものである.

1.4 関連研究

本研究と同様に, 通信プロトコルに関連するプログラムを, 形式手法を用いてバグのないように実装する研究としては, EverParse: Verified Secure Zero-Copy Parsers for Authenticated Message Formats[8] がある. EverParse は,DSL 形式仕様言語からメモリ安全性が保証されたパーサを生成するためのフレームワークであり,DSL 形式仕様記述言語コンパイラである QuackyDucky と, パーサを生成する LowParse と呼ばれる 2 つの部分で構成されている. この QuackyDucy のメモリ安全性は保証されていないが, パーサを生成する LowParse は F*言語によりメモリ安全性が保証されている. この関連研究ではこのような特徴を持つ EverParse を利用し認証メッセージ向けのパーサを生成し, そのパーサの性能評価を行っている. EverParse は DSL 形式仕様言語により定義したデータフォーマットもとにセキュアなパーサを生成するため, 利用に際して DSL 形式使用言語の学習コストがかかり, また汎用性にも欠ける. これに対し, 本研究で開発した MQTT パケットパーサは F*言語のみを用いて開発しているため, 利用は F*言語のみの学習コストで済み, 汎用性も高い.

1.5 プログラム検証

1.5.1 F*言語による開発の前提条件

F*言語を用いた開発において脆弱性が無いことを保証するためには, 開発において以下のような前提条件が必要である.

(1) プログラムの関数レベルでの仕様書の内容が正しいこと
: F*言語のプログラム検証は, 関数の入出力に対する制約条件を基に行われる. このため, 少なくとも関数の機能レベルでのプログラム仕様が正しいことが F*言語によるプログラム検証が正しく動作する前提となる.

(2) F*による仕様記述は, 仕様書の内容と乖離がないこと
: (1) を満たすためには, 開発者が仕様書を正しく解釈し

© 2021 Information Processing Society of Japan

た上で, それを正確に F*言語で記述する必要がある.

2. インタビュー

本研究ではまず, F*言語を用いたプログラム開発の利点や困難さを明らかにするため, F*言語の基礎知識を持つ 3 人を対象としてインタビューを行った.

2.1 インタビュー協力者について

インタビュー協力者 3 名の共通点は以下の通りである.

- プログラミング経験者である.
  全員 5 年以上のプログラミング経験があった.
- セキュリティに関する基礎的な知識を持つ.
  例えば, バッファオーバーフローの知識や, その対策法に関する知識を持っていた.
- 開発で主に利用するプログラミング言語は C 言語である.
  プログラム開発で利用するプログラミング言語の種類に関しての質問を行ったところ, インタビュー参加者全員が C 言語をよく利用するとの回答を得た.

2.2 インタビュー内容

インタビュー形式は半構造化インタビューとし, 一人あたり約 30 分をかけた. そのインタビュー協力者 3 人のインタビュー結果を統合し 内容分析を行った結果, 大きく分けて'F*言語の文法難易度', 'F*言語の保守性', ' 関数の制約条件の妥当性検証' の 3 つのカテゴリに関する分析結果を得た.

2.2.1 文法難易度

インタビュー対象者からは, "F*言語は個々人が学んでいたプログラミング言語の中で, 難しい部類に入るプログラミング言語" という共通の回答を得た. なぜ F*言語の文法は難しいと感じさせているのかインタビュー内容から分析した結果' 関数型言語' と, ' 関数の制約条件' が F*言語の文法を難しく感じさせる観点である判明した.

関数型言語について インタビュー内容を分析した結果, "F*言語以外の関数型言語での開発経験があれば, F*言語の文法自体に対する難易度は既に開発経験がある関数型言語と比較するとそこまで高いものではない" とする意見や, "関数型言語での開発経験が無いと手続き型言語とのプログラムの書き方の違いに起因して, F*言語全体の文法を難しいと感じる" という共通した意見が得られた.

関数の制約条件について インタビュー内容を分析した結果, 関数型言語の開発経験の有無に関わらず, F*言語の中で難しく感じる文法に関数が満たすべき制約条件の記述があった. 回答の一つを挙げると, ' そもそもどのような方針で, どこの部分を, どの程度の規模で関数が満たすべき制約条件の記述をすればよいかの検討が難しい' という意見があった.

2.2.2 F*言語の保守性

' 関数の制約条件', ' プログラム修正' の観点でインタビュー内容を分析した結果から, F*言語で開発を行ったプログラムは, 一般的なプログラミング言語と比較して, 保守性が高いと評価されていることが明らかになった. これは,F*言語は不具合が許されない分野でのシステム開発に役立つことを示唆している.

関数の制約条件について F*言語の仕様記述に関する質問の回答として, "C 言語との比較であれば, F*プログラムの保守性は高く, そして関数の制約条件を見ればプログラム全体の動作の流れがすぐにわかる" ということや, "求められるプログラムの仕様が変更になったとしても関数の制約条件のおかげでプログラムの修正箇所がすぐに把握できる" という意見を得た.

プログラム修正について F*言語はどんな人に向いているかという質問の回答として, "F*言語のプログラムをあとから効率的に修正するためには, F*言語の開発経験が必要" という回答や, "F*言語を正しく活用できるのは, 技術的な基礎知識があり, 技術を使うだけではなく, 技術の仕様を正しく意識し, 設計, 開発ができる人" という回答が得られた. 特に後者の回答は, F*言語を十分に活用できる可能性がある人材が限定されることを示唆している.

2.2.3 関数の制約条件の妥当性検証

既に述べたように F*言語によるプログラミングが仕様を正しく反映するためには関数の制約条件が仕様を正しく反映していることが求められる. しかし, インタビュー内容の分析結果からコードレビューも, ソフトウェアテストも F*プログラム中の関数の制約条件と, 実際の求められているプログラムの仕様との乖離を減らすことはできても, 乖離がなくなったことの保証はできないと評価されていることが判明した.

関数の制約条件の正しさについて F*プログラム中の関数の制約条件と, 実際の求められているプログラムの仕様との乖離をどのように無くしていけば良いと思うかという質問の回答として, "現実的な解決策の一つが, 複数人での関数の制約条件の正しさを確認することではないか" という回答や"F*言語の検証機能に加えてソフトウェアテスト (Fuzzing なども含む) も 2 重に活用するのが良いのではないか" という回答が得られた.

3. 開発

MQTT Version 5 のパケットのデータ構造は, パケットタイプに関わらず同様のデータ構造を持つ固定長ヘッダ ('Fixed Header'). そして, 固定長ヘッダの直後に存在しパケットタイプによりデータ構造が変化する可変長ヘッダ ('Variable Header'). 可変長ヘッダ以降は, ペイロード

© 2021 Information Processing Society of Japan

('Payload') と呼ばれている．ただし, このペイロードはパケットタイプにより存在しない場合もある．本研究ではこの, MQTT のパケットパーサを F*言語で開発した．そのフローチャートは, 図 2 のようになった.

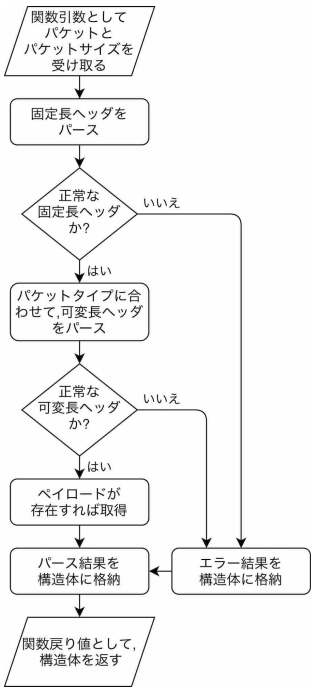

図 2 パケットパース関数のフローチャート

3.1 関数仕様

今回 F*言語で開発した MQTT パケットパース関数のプロトタイプ宣言は, Listing1 の通りである.

Listing 1 パース関数のプロトタイプ宣言 (C 言語)

```
1
2  typedef struct parse_result_s
3  {
4    uint8_t message_type;
5    C_String_t message_name;
6    struct_flags flags;
7    uint32_t remaining_length;
8    struct_connect connect;
9    struct_publish publish;
10   struct_disconnect_reason disconnect;
11   struct_property property;
12   struct_error_struct error;
13 }
14 parse_result;
15
16 parse_result mqtt_packet_parse
17 (
18       uint8_t *packet_data,
19       uint32_t packet_size
20 );
```

関数名 mqtt_packet_parse

第 1 引数 (packet_data) パースを行いたい MQTT Version 5 のパケットデータ

第 2 引数 (packet_size) パース対象のパケットデータのサイズ

戻り値 (parse_result) パース結果

パース結果が格納されている戻り値 (parse_result) のデータ構造は以下の通りである.

message_type MQTT Version 5.0 の仕様書中の'2.1.2 MQTT Control Packet type' の 'value' と対応しており, この値を見ればパースしたパケットの種別がわかる．例えば, この値が 1 であれば CONNECT パケットであるとわかる.

message_name message_type に対応したパケット名が文字列として格納されている．message_type が 1 のとき, 格納されている文字列は 'CONNECT' である．'C_String_t' は 'const char *' 型と同等である.

flags MQTTv5 の固定ヘッダには, 'flag' と呼ばれるものが存在する．仕様書の'2.1.3 Flags' にその記述がある．'flag' に関する情報が格納されている構造体である.

remaining_length これは, '2.1.4 Remaining Length' に記述がある．MQTTv5 パケット全体のサイズから, パケットの種別に関わらず共通の固定長ヘッダサイズを減算した値である．これは, 可変長ヘッダと, それ以降に続くペイロード部のサイズを示す.

connect MQTTv5 の Connect パケットをパースした結果が格納されている．message_name が'CONNECT' の場合は, この構造体に CONNECT パケットのパース結果が格納されている.

publish MQTTv5 の Publish パケットをパースした結果が格納されている．message_name が'PUBLISH' の場合は, この構造体に PUBLISH パケットのパース結果が格納されている.

disconnect MQTTv5 の Disconnect パケットをパースした結果が格納されている． message_name が'DISCONNECT' の場合は, この構造体に DISCONNECT パケットのパース結果が格納されている.

property MQTTv5 の可変長ヘッダには, 'Properties' と呼ばれているデータ部が存在する． これは, 仕様書の'2.2.2 Properties' に記述がある．この構造体には, その'Properties' に関するパケットパース結果が格納されている.

error パケットパース時のエラー発生有無を確認するために用いる構造体である． この構造体は, メンバ変数として 'code' と'message' を持つ． code は mqtt_packet_parse 関数の終了ステータスの役割を担っている．code が 0 の場合は, パケットのパースが正常に終了したということであり, message には空文字

© 2021 Information Processing Society of Japan

列が格納される．code が 0 以外であれば，エラーに対応したエラーメッセージが格納されている．そのため，parse_result の結果を確認する際は一番初めに，parse_result.error.code が 0 か非 0 かの確認を行う必要がある．

mqtt_packet_parse 関数には，満たすべき関数の事前条件が存在する．パケットパース関数が，以下の条件を満たさない値を引数として受け取った場合のプログラム動作は未定義である．

- packet_size は 268435461 以下であること
- packet_data が null ではないこと
- packet_data の領域サイズと，packet_size が等しいこと

4. 評価

本研究で実装した MQTT パケットパーサを形式手法により検証する時間，および MQTT パケットパーサそのものの性能と安全性を評価した．

4.1 計測環境

本評価は，以下のような環境構成で行った．

4.1.1 実機

OS

Ubuntu 20.04.1 LTS

アーキテクチャ

x86_64

CPU

Intel(R) Core(TM) i7 CPU 970 @ 3.20GHz

CPU スレッド数

12

メモリ容量

8.00GB

4.1.2 Docker

Docker Engin バージョン

19.03.13

CPU コア使用上限数

12

メモリ使用量上限数

8.00 GB

4.1.3 Docker Image

OS

Ubuntu 18.04.4 LTS

アーキテクチャ

x86_64

コンパイラ

gcc version 7.5.0 (Ubuntu 7.5.0-3ubuntu1 18.04)

F*

F* 0.9.7.0-alpha1

platform=Linux_x86_64

compiler=OCaml 4.05.0

Kremlin バージョン

0.9.6.0

4.2 プログラム検証時間

本研究で開発した MQTT パケットパーサーの F*言語によるプログラム検証時間の計測を行った．プログラム検証時間の計測は，同一ソースコードに対する検証を 10 回行いその検証時間のみ (コンパイルや実行を除く) を time コマンドにより行った．計測結果は図 3 の通りになった．

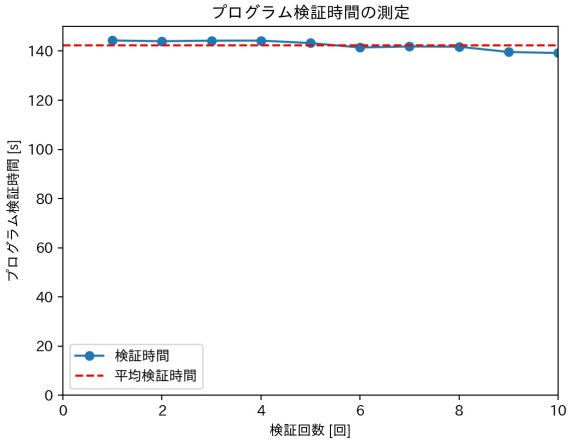

図 3 プログラム検証時間の計測結果

プログラム検証時間には多少時間のばらつきが認められるが，同一ソースコードに対する検証時間に大きなばらつきはなかった．平均プログラム検証時間は，142.4005s であった．

4.3 性能

4.3.1 既存の MQTTv5 パケットパーサとの比較

既存の MQTT Version 5 対応のパケットパーサ機能を持つ実装として，”emqtt5”と呼ばれるオープンソース MQTT ライブラリがあるため，本研究での実装と比較した．以下，区別のため本研究で開発した MQTT パケットパース関数を verimqtt と呼ぶ．emqtt5 は，MQTT パケットパーサ関数にパケットデータとそのサイズを与えると，整形されたパース結果文字列を返す．一方，verimqtt は，パケットデータとそのサイズを与えると，渡した MQTT パケットのパース結果が格納された構造体を返す．このように，verimqtt，emqtt それぞれのパケットパース関数が返す返り値が異なるため，性能比較をできるかぎり同等の条件で行うために verimqtt を利用して emqtt5 パケットパーサの機能をソフトウェア的に再現した．このように条件を揃えた上で，verimqtt と emqtt5 の性能を’ パケット 1 個あたりの平均パース時間’ の観点で比較した．ただし，計測条件をできる限り揃えるために以下のような制約を満たす計測環境を整

© 2021 Information Processing Society of Japan

えた.

同一実行ファイル上でパース時間の計測を行う

emqtt5 が C++言語製, verimqtt が C 言語製とそれぞれの開発言語が異なるため, 開発言語により生じる差異などをできる限りなくした. また, これに伴いそれぞれのパーサライブラリに対するパケットデータの渡し方も共通化した.

パケットパース結果の標準出力は行わない

emqtt5 と, verimqtt が持つパース機能はほぼ同等であるが, それぞれのパーサが持つパース結果の標準出力はパース時間の計測には不要であるため計測プログラムから除いた.

計測方法は以下の通りである. 事前に用意した MQTT 内での役割が異なる 16 種類の MQTT パケット, それぞれを emqtt5, verimqtt それぞれで 10 万回パースしパケット 1 個あたりの平均パース時間の計測を行った. ここでのパケット 1 個あたりの平均パース時間とは, emqtt5, verimqtt それぞれのパケットパース関数に 1 つパケットを与えた時から, それぞれの関数が同様のパース結果文字列を返すまでの平均経過時間である. また, verimqtt のパケットパース関数に 1 つのパケットを与えた時から, パース結果が格納された構造体を返すまでの平均経過時間も計測した. コンパイラの最適化レベルにより, パケットパース時間に大きな変動が起きないか確認するために, 最適化レベル 0 の場合 ('-O0') と, 最適化レベル 2 の場合 ('-O2') での計測を行った. MQTT Version 5 パケット 16 種類の内訳は, Connect パケット 5 種類, Disconnect パケット 2 種類, Publish パケット 9 種類 (7 種類が異なるデータ型を持つプロパティ属性) であり, パケットサイズもそれぞれ異なり, 16 種の平均パケットサイズは 31.625 バイトであった. この条件下での計測結果は以下の通りになった.

平均パース時間 (最適化レベル 0)

emqtt5(パース結果文字列を返すまで) 7.053 μs
verimqtt(パース結果文字列を返すまで) 28.830 μs
verimqtt(パース結果構造体を返すまで) 2.599 μs

平均パース時間 (最適化レベル 2)

emqtt5(パース結果文字列を返すまで) 5.119 μs
verimqtt(パース結果文字列を返すまで) 24.375 μs
verimqtt(パース結果構造体を返すまで) 1.834 μs

この評価では emqtt5 と同様の整形されたパース結果文字列を返す機能を, verimqtt パケットパーサを用いて再現したが, verimqtt の方が emqtt5 より実行時間が大きくなった. だが, 本研究で開発した verimqtt がパース結果構造体を返すまでの時間と, emqtt5 がパース結果文字列を返す時間とを比較すると verimqtt の計測時間のほうが短時間で済んでいた. この計測時間の差は, コンパイラの最適化レベルによって変わることはなかった. この計測結果から, verimqtt は パケットパース結果を格納した構造体を返すため, パケットパースライブラリとしての汎用性は高いがパケットパース結果を整形されたパース結果文字列を構成するためだけの目的には向かないと考えられる. また, 本研究では F*言語を用いて C 言語プログラムを作成したが, それがプログラムの性能に大きな悪影響を与えないことがわかる.

4.3.2 Fuzzing テスト

本研究での実装に脆弱性があるかどうかを評価するため, verimqtt に対し 7 日間 American Fuzzy Lop[9] による Fuzzing テストを行った. American Fuzzy Lop(以下 AFL) は, オープンソースの Fuzzing ツールであり, 遺伝的アルゴリズムを活用し, プログラムの実行経路の探索を効率よく行っている. AFL による評価の結果, verimqtt はプログラムのクラッシュ, ハングともに発生せず, 図 4 のような実行結果となった. すなわち, verimqtt には脆弱性は容易にはみつからず, 堅牢性が確保されていると考えられる.

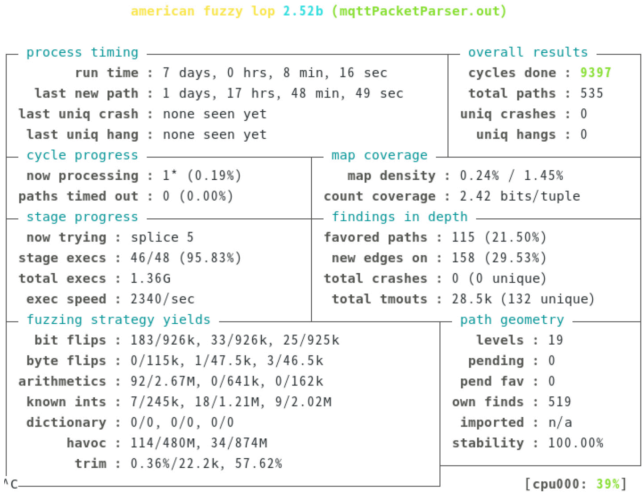

図 4 7 日間 AFL による Fuzzing を行った結果

4.4 ソースコードサイズ

verimqtt を実装した F*プログラムのソースコード行数の内訳は, コード行数 4454 行, コメント行数 491 行, 空行数 304 行となった. この, F*ソースコードから自動生成された C ソースコード行数の内訳は表 1 のとおりとなった. また, gcc コンパイラの最適化レベルが 0(-O0) のときの実行ファイルサイズは 222K バイトであった.

表 1 F*コードから自動生成された C ソースコード

ファイル種類	空行数	コメント行数	コード行数
C ファイル	453	48	4505
C ヘッダ	571	72	1238
合計行数	1024	120	5743

© 2021 Information Processing Society of Japan

5. 開発の過程で明らかになった課題

本研究で MQTT パケットパーサの開発を行う過程で, 明らかになった主な課題は 2 つある. まず一つ目は, F*言語でプログラム開発を行う者にとって正しく関数の引数と返り値がそれぞれ満たすべき制約条件の記述を行うことが難しいことである. 実際, 我々が MQTT パケットパーサの動作の確認を行っていた際に制約条件の記述の誤りに気づいたことがあった. そのため対策として, MQTT パケットパーサの開発にソフトウェアテストも導入し F*言語の検証とのダブルチェックを行うようにした.

二つ目は, F*言語を用いたプログラム開発のコストが大きいことである. MQTT には 15 種のコントロールパケットが定義されているが, 約 1 年間の MQTT パケットパーサ開発でパースが可能になったのは重要な 3 種のパケットのみであった. これは当初想定していた開発機関よりかなり長い. 本研究で行ったインタビュー内容の分析結果としても, F*言語でのプログラム開発のコストは大きいという結果が得られた.

6. おわりに

本研究では, 堅牢なプログラム開発手法導入の妨げとなっている課題を明らかにする目的で, プログラム検証用言語である F*言語を用いて MQTT パケットパーサの開発と性能評価, 安全性評価を行った. その結果, F*言語を用いることが堅牢なシステムの開発に役立つという結果は得られたが, 同時に開発における課題が明らかになった. すなわち F*言語でプログラム開発を行う者にとって正しく関数の引数と返り値がそれぞれ満たすべき制約条件の記述を行うことが難しいことや,F*言語の文法が一般的なプログラミング言語と比較して難しいことである. これらは, F*言語を用いた形式手法の普及の阻害要因であると考えられるので, 今後はこの普及の阻害要因を取り除く手法について研究を進めていきたい.

謝辞

本研究の一部は, アドソル日進株式会社からの受託研究によるものである. また, 本研究のインタビューに協力していただいた匿名の大学生 3 名に感謝いたします.

参考文献

[1] 脆弱性対策情報データベース JVN iPedia の登録状況 [2019 年第 2 四半期（4 月〜6 月）](online), 入手先 ⟨https://www.ipa.go.jp/security/vuln/report/JVNiPedia2019q2.html⟩ (閲覧日: 2019.07.24).
[2] F*言語 [Introduction, People](online), 入手先 ⟨https://fstar-lang.org/⟩ (閲覧日: 2021.01.06).
[3] Project Everest[The HTTPS Ecosystem , A combination of several sub-projects](online), 入手先 ⟨https://project-everest.github.io/⟩ (閲覧日: 2021.01.06).
[4] miTLS: A Verified Reference Implementation of TLS [Introduction](online), 入手先 ⟨https://mitls.org/⟩ (閲覧日: 2021.01.06).
[5] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, Benjamin Beurdouche: *HACL*: A Verified Modern Cryptographic Library*, Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications pp.1789–1806(2017).
[6] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath Setty, Laure Thompson: *Vale: Verifying High-Performance Cryptographic Assembly Code*, Proceedings of the USENIX Security Symposium p.917-934(2017).
[7] MQTT Version 5.0 OASIS Standard(online), 入手先 ⟨https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html⟩ (閲覧日: 2019.03.07).
[8] Tahina Ramananandro, Antoine Delignat-Lavaud, Cédric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, Jonathan Protzenko: *EverParse: Verified Secure Zero-Copy Parsers for Authenticated Message Formats*, Proceedings of the 28th USENIX Conference on Security Symposium pp.1465-1482(2019).
[9] american fuzzy lop (2.52b)[Introduction](online), 入手先 ⟨https://lcamtuf.coredump.cx/afl/⟩ (閲覧日: 2021.01.06).

© 2021 Information Processing Society of Japan