同一コード実行回数を利用したファジングの検査密度向上手法

青木 慧[†1]　中西 福友[†1]　春木 洋美[†1]

概要：制御システムのセキュリティ評価では，システムの可用性を検証するために，ファジングと呼ばれる検査手法が用いられる．プログラムの各所に点在する脆弱性を見つけるためには，同一コードを様々な検査値で検証して，その検査密度を向上させる必要がある．しかし，一般的なファジングツールでは，大量のテストケースからランダムに選択して送ることから検査密度が低い箇所が生じる．一方で，検査密度を上げるために，全組み合わせを網羅した検査を行うことは検査時間が膨大になるため，現実的ではない．本稿では，テストケースと検査対象の実行回数履歴とを対応付け分析することで，少ない追加検査数で検査密度を向上させる手法について論じる．

キーワード：制御システムセキュリティ, 脆弱性検査, ファジング, 追加検査

Fuzzing Method for Improving Test Density using the Execution Degree of Source Code

Satoshi Aoki[†1]　Fukutomo Nakanishi[†1]　Hiroyoshi Haruki[†1]

Abstract: In order to verify the availability of the system, fuzzing is often used in the security evaluation of the industrial control system. In vulnerability test, it is important to scan vulnerabilities in various places of the program as far as possible. An effective method is to improve test density with testing the same code by various values. However, a general fuzzing has a problem that there are low test density places in the program. This is caused by random selection from many generated test cases. On the other hand, the test method that covers all patterns in order to maximize the test density is impossible in terms of the feasible time. In this paper, we discuss how to improve the test density in additional test cases using the execution degree corresponding to each test case.

Keywords: Industrial Control System Security, Vulnerability Test, Fuzzing, Additional Test

1. はじめに

1.1 背景

従来, 産業用制御システム（Industrial Control System: ICS）は，特有のプロトコルや OS を利用し，かつ，クローズドネットワーク内に閉じていたため，サイバー攻撃とは無縁と考えられていた．しかし，2010 年にイランで起きた核燃料施設の制御システムがコンピュータウイルス Stuxnet に感染した事例から，外部のネットワークと切り離して運用していたとしても USB メモリなどの外部メディアを経由してウイルスに感染することが広く認識された．ウイルスに感染することで，可用性を重視するために今まで放置していたシステム内部に残存する脆弱性が悪用されてしまう．そして, 制御システムの脅威となるのは USB メモリだけではない．独立行政法人情報処理推進機構（IPA）が 2015 年度に制御システムユーザ企業を対象に行った調査によると，リモートメンテナンス回線への対策として,「利用時のみ接続させる」が 46.0%,「接続させる端末の認証を行う」が 39.0%と，多くの制御システムユーザがリモートメンテナンス回線を利用していることが分かる[1]．リモートメンテナンス回線は，攻撃者が利用できるアクセス手段の一つであり，常に通信状況を監視しておくなどのセキュリティを考えた運用を行っていたとしても，制御システム事業者にとっては脅威であることに変わりはない．さらに，近年では制御システムでも IoT（Internet of Things）が普及しつつある．ゆえに，オープンな通信プロトコルや OS の採用が増え，今までクローズドとされてきた制御機器もより汎用ネットワークに接続され，悪意のある攻撃を受ける機会がより増えると予想される．

2014 年 11 月にサイバーセキュリティ基本法が成立し，2015 年 1 月には内閣にサイバーセキュリティ戦略本部が，内閣官房に内閣サイバーセキュリティセンター（National center of Incident readiness and Strategy for Cyber security: NISC）が設置されるなど，高まるサイバー攻撃から国民生活や経済活動を守るための国家的な取り組みが行われている．サイバーセキュリティ基本法に基づき定められたサイバーセキュリティ戦略では，システムには脆弱性が内在しているものであるという現実を認識したうえで，システムの企画・設計段階からセキュリティの確保を盛り込むとともに，提供する様々な製品・サービスについてセキュリティ面でも適切な説明を行うことや，システム運用時には関係者が連携してシステムを構成する機器などの脆弱性を調査し，ベンダに対して修正を促すとともに，事業者に着実

†1 株式会社　東芝
Toshiba Corporation

に対策が行き届くような仕組みを検討し，構築していくことの必要性が記されている[2].

また，サイバーセキュリティ戦略では，制御系システム等の調達，運用に高度な専門性が必要とされることから，セキュリティ要件への適合を客観的に判断することが可能である国際標準に則した第三者認証制度の活用が進められている．制御システムのセキュリティ認証としては，Wurldtech 社が提供するファジングツール Achilles を利用した Achilles 認証を皮切りに，国際標準規格である IEC 62443 に準拠した形で様々な認証が立ち上がっている．制御システムに関する組織のマネジメントシステムに対する認証である CSMS（Cyber Security Management System）認証は，IEC 62443-2-1 に準拠し，日本では一般財団法人日本情報経済社会推進協会（JIPDEC）が認定機関となって運用を行っている．制御システムを構成する制御機器に対する認証である EDSA（Embedded Device Security Assurance）認証は，IEC 62443-4-2 に準拠し，スキームオーナーである ISA Security Compliance Institute（ISCI）との協力により，日本では日本適合性認定協会（JAB）が認定機関に，技術研究組合制御システムセキュリティセンター（CSSC）の CSSC 認証ラボラトリーが認証機関となり運用を行っている[3].

EDSA 認証では，Programmable Logic Controller（PLC）やフィールドセンサデバイス，Distributed Control System（DCS）コントローラなどの組み込み機器を認証の対象としている．EDSA 2.0.0 と呼ばれるバージョンアップした EDSA 認証では，主に 3 つの項目を評価・試験することで想定脅威への対策が十分であることを認証している[4]-[7]. 1 つ目は，体系的な設計不良の検出・回避のための評価，2 つ目は，実装エラーや実装漏れを検出するための評価，最後は，デバイスの堅牢性を評価する試験である．これらのうち，EDSA 認証におけるデバイスの堅牢性を評価する試験は，Embedded device Robustness Testing（ERT）と呼ばれ，大きく 2 つの試験に分かれている．1 つ目の試験は Vulnerability Identification Testing（VIT）と呼ばれるもので，既知の脆弱性の存在を検査するものである．2 つ目の試験は Communication Robustness Testing（CRT）と呼ばれるもので，デバイス内部の潜在的なセキュリティ脆弱性の存在を検査するものである．VIT に関しては，既知の脆弱性データベースを持ったスキャンツールを利用して制御機器内部における既知脆弱性の有無をチェックするという目的のため，検査方法としては制御機器で利用されているアプリケーションのバージョンをチェックするなどであり難しくない．しかし，CRT に関しては，未知の脆弱性の存在を検査するものであるため，自ら脆弱性の定義を行い，異常なトラフィックに対して脆弱でないことを検査することが目的である．そのため，どのような異常なトラフィックをどれだけ与えれば堅牢であるかといった検査の十分性を明確に示すことが難しいといった課題がある．現状は，検査の十分性を EDSA 認証が認定したファジングツールの性能に委ねている．

1.2 目的

本手法が解決しようとする課題は，プログラムに含まれる十分に検査されていない要素を検査対象機器に実行させることができるテストケースの条件を決定することにある．検査対象機器がテストケースによって十分に実行されていない要素は，検査が十分に行われていないという考えに基づいている．また，1 つの要素に対して様々な検査値で検査できているかという検査密度の観点において，要素単位で検査が十分であることを示す 1 つの指標に実行回数が有用であるかを評価する．

2. ファジング

2.1 用語定義

本稿では，検査用に生成したテストケースと，実際に検査で利用するテストケースとを区別して説明するため，前者のテストケースをテストデータと呼び，後者のテストケースを入力データと呼ぶ．

2.2 概要

未知脆弱性を検査する前述の CRT には，ファジングと呼ばれる手法が用いられる．ファジング手法の概要を図 1 に示す．まず，検査対象が正常な通信を行う際に受け取るデータ（正常データ）をもとにアノマリやファズと呼ばれる異常なデータ部分を含んだテストデータを作成する．次に，生成したテストデータのうち検査対象に対して送るデータ（入力データ）を選択し，大量に送る．そして，検査対象からの応答や挙動などを監視することによって脆弱性を検出する．正常に応答が返ってきた場合は，直前に送った入力データで脆弱性が発現しなかったとして PASS を示す．一方，応答が返ってこなかった場合は，直前に送った入力データで脆弱性が発現したとして FAIL を示す．

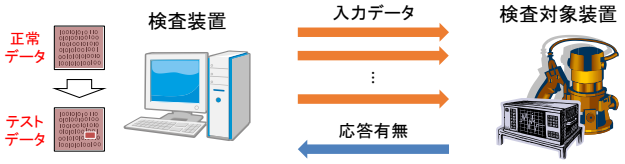

図 1　ファジング手法の概要

2.3 テストデータの生成方法

検査装置では，テストデータを大量に生成する．テストデータの生成方法には大きく分けて 3 種類の方法がある[8][9]．Dumb Fuzzing と Mutation-based Fuzzing と Generation-based Fuzzing である．3 種類のテストデータ生成方法の特徴を表 1 に示す．Dumb Fuzzing は，検査対象が利用しているデータの構造解析や正常データを利用せず，

データ長やデータ値を全くランダムにテストデータを生成する方式である．そのため，生成するのは簡単であるが，検査の効率が悪く応用性も低い. Mutation-based Fuzzing は，データ構造の解析は行わないが，正常データをもとにデータ値の一部を変異させてテストデータを生成する方式である．Generation-based Fuzzing は，RFC などの技術規格や正常データからデータ構造の解析を行い，定義ファイルを作成する．そして，定義ファイルをもとにデータ構造の要素（例えば，ネットワークプロトコルのフィールドや，画像ファイルのセグメントなど）別にランダムなデータ長やデータ値のテストデータを生成する. 定義ファイルを作成し，データ構造の要素別にテストデータを生成するため，作成に手間がかかるが，特定の構造部分のみに特化させた検査や正常なデータ構造と照らし合わせた時に異常と判定されるようなテストデータを意図的に生成できるなど検査のアレンジをしやすいという特徴がある．

表 1　テストデータの生成方法

	Dumb Fuzzing	Mutation-based Fuzzing	Generation-based Fuzzing
データ構造解析	利用しない	利用しない	利用する
正常データ	利用しない	利用する	どちらでも
生成しやすさ	◎	○	×
応用性	×	○	◎

2.4 入力データの送信方法

検査装置では，入力データを検査対象装置に送信する．ファジングにおける入力データの送信方法には大きく分けて 2 種類の方法がある．1 つは大量の入力データを同時に検査対象に送る方法で，検査時間が短いというメリットがあるが，脆弱性があった場合には再現可能な入力データを絞るために追加検査の必要があるなどデメリットがある．もう一方は入力データを 1 つずつ順次に検査対象に送る方法で，脆弱性を発現させた入力データを特定しやすいというメリットがあるが，検査対象の状態確認のために応答を待つ時間が入力データごとに発生するので検査時間が長いというデメリットがある．

2.5 実行履歴を利用したファジング

実行履歴を利用したファジング手法の有用性については文献[10]にて示唆されている．図 2 に実行履歴の一例を示す．図 2 は，プログラムの要素をコード毎に分割し，コード A は 100 回，コード B は 0 回といったように，コード毎に実行回数を記録したものである．一般にコード網羅率というと，プログラム全体のコード数に対して 1 回以上実行されたコード数の割合のことを言い，図 2 の場合はコード網羅率が 80%であるといえる．文献[10]では，コード網羅率は検査結果を分析する上で重要な概念であり，実施した検査の信頼性を定量的に示す指標として利用できるとしている．実際に，単体テストなどではソフトウェアの品質向上のためにカバレッジ（網羅率）を測定及び分析し，テストが実施されていないコードを確認することによって，テストの妥当性を向上させている．また，脆弱性検査分野においてもファジングをしながらコードカバレッジを計測することで，コードカバレッジの高低で入力データを評価でき，遺伝的アルゴリズムを利用して新しいテストデータを生成するファジング方法が提案されている[11].

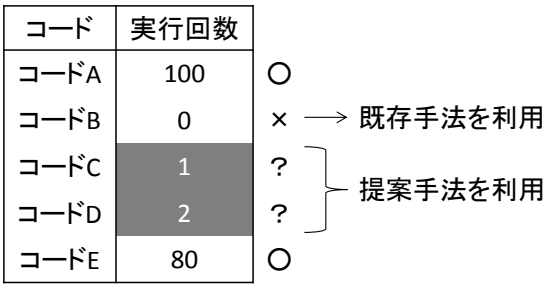

図 2　実行履歴の一例

2.6 課題

ファジングでは，大量のテストケースをランダムに生成・選択するので，検査対象装置が検査中に実行するプログラムの要素に偏りが生じる場合がある．図 2 の場合で説明すると, コード A は 100 回実行されているにも関わらず, コード C は 1 回しか実行されなかった，といった場合がある．検査において実行された要素に偏りが生じた場合，開発したプログラムは，部分的に検査が十分でない要素を含むため，その要素が脆弱性を有している可能性を排除しきれない．なお，検査が十分でない要素には 2 種類の考え方がある．1 回も実行されていない要素と 1 回以上実行された要素である．図 2 のコード B のように，1 回も実行されていない要素を減らす方法については，検査網羅率の観点から文献[9]や文献[11]のように既に取り組まれている．しかし，図 2 のコード C 及び D のように，1 回以上実行された要素について，より効率的に検査密度を高める方法についてはは取り組みがない．

検査対象装置が検査中に実行するプログラムの要素に偏りが生じないようにするには，検査において，全ての組み合わせを網羅したデータを生成し，生成した全てのデータを対象装置に供給して応答および挙動を監視すればよい．例えば，1 バイトの入力しか受け付けない検査対象装置の場合には，256 個のテストケースを生成し，検査を行えばよい．しかし，このような方法ではテストケース数が膨大となってしまうため，現実的ではない．

3. 提案手法

3.1 概要

本手法は，一次検査を行った際の入力データと実行履歴を利用することによって，検査が不十分である要素（未達要素）を特定する．そして，未達要素を実行した入力データに類似する入力データを選択することによって，二次検査以降では未達要素を集中的に検査可能にする手法である．ここでいう検査が不十分であるとは，プログラム要素の実行回数が一定回数以下などの，実行履歴が検査基準を満たしているか否かを指す．

図 3 に本手法の検査フローを示す．本手法では，予め 1 つの入力データに対して 1 つの実行履歴を出力できるようにしておく．まずは，ランダムにテストデータの生成を行う．生成したテストデータの中から一次検査に利用する入力データの初期選択を行う．検査時には，入力データそれぞれの実行履歴を出力する．次に，これらの入力データと実行履歴とを分析し，未達要素を特定する．未達要素がある場合には，入力データと実行履歴に基づいて未達要素を実行可能な入力データを追加選択する．そして，二次検査として選択した入力データを用いて検査を実施する．つまり，未達要素を実行した入力データを利用して類似する新たな入力データを追加選択することで，未達要素を効率よく検査することができる．

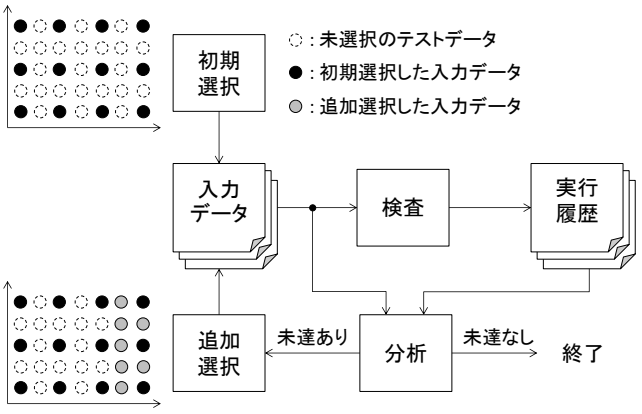

図 3　提案手法の検査フロー

3.2 未達要素の特定

本手法では，一次検査を行った際の入力データと実行履歴を利用することによって，未達要素を特定する．未達要素が存在しない場合は，どの部分においても検査密度が十分であると判断し，以降の追加検査を行わない．

まず，検査に使用した入力データと検査時に生成した個別実行履歴の対応付けを行う．また，個別実行履歴を合成して，総実行履歴を生成する．例えば，プログラムに含まれる命令文ごとに，複数の個別実行履歴のそれぞれから対応する実行回数を取得する．そして，プログラムに含まれる命令文ごとに，取得した複数の実行回数を加算することにより，総実行履歴を生成する．これらをまとめたものを実行情報と呼び，図 4 に一例を示す．入力データ毎に検査対象であるプログラムのどのコードを何回実行しているか（個別実行履歴）をまとめ，最下行は総実行回数（総実行履歴）を示している．具体的には，識別情報「001」の入力データは，行番号「01」のコードを 10 回，行番号「02」のコードを 0 回，行番号「03」のコードを 0 回，行番号「04」のコードを 1 回実行したことが個別実行履歴から分かる．また，行番号「01」のコードは 7 個の入力データによって計 28 回実行されていることが総実行履歴から分かる．

	入力データの識別情報	コード行番号			
		01	02	03	04
入力データ毎の個別実行履歴	001	10	0	0	1
	002	2	0	0	1
	003	2	0	0	1
	004	2	1	0	1
	005	2	1	0	1
	006	5	0	0	1
	007	5	0	0	1
総実行履歴	計	28	2	0	7

図 4　実行情報の一例

次に，検査対象装置が実行するプログラムに含まれる複数の要素のうち，総実行履歴が検査基準を満たしていない未達要素を特定する．例えば，図 4 の場合で検査基準を「実行回数が 7 以上」とした時，総実行履歴に示された実行回数が検査基準を満たさないコード，つまり行番号が「02」のコードを，未達要素として特定する．一方で，実行回数が 0 回の要素，つまり行番号が「03」のコードについては，実行した入力データが存在せず利用できないため，本手法では未達要素として扱わない．

3.3 追加検査用入力データの選択

特定した未達要素を集中的に検査するために，未達要素を検査対象装置に実行させることができる入力データを選択する．未達要素が複数ある場合にはそれぞれの未達要素に対して追加検査用入力データの選択を行うが，本稿では説明を省略する．

まず，複数の入力データのそれぞれを，特定した未達要素を検査対象装置に実行させたデータ（実行有りデータ）と，未達要素を検査対象装置に実行させていないデータ（実行無しデータ）とに分類する．例えば，図 4 の場合には未達要素である行番号「02」に対しては，実行有りデータは識別情報「004」と「005」の入力データであり，実行無しデータは未達要素を 1 回も実行していない残りの「001」，「002」，「003」，「006」及び「007」の入力データというよ

うに分類できる．

次に，実行有りデータと実行無しデータとの関係に基づき，実行有りデータと共通の特徴を有する入力データの条件を決定する．例えば，実行有りデータの集合の属性を表すパラメータの条件を決定する．具体的には，何れかのパラメータについて，実行有りデータと実行無しデータとの境界値を決定する.そして,パラメータの境界値に基づき，パラメータの条件を決定する．図 5 は，パラメータ#1 とパラメータ#2 についてテストケースをマッピングし，実行有りデータと実行無しデータとの境界条件の一例を示した図である．Generation-based で生成されたテストデータは正常データを中心に 1 つのパラメータの値を変異させてつくられることが多く，例えば，図 5 のように十字型にマッピングできる場合がある．複数のパラメータの値を変異させたテストデータや，それ以外の観点で生成されたテストデータも実際にはあるが，本稿では説明を省略する．図 5 では，実行有りデータの集合は，パラメータ#1 が α より大きく，実行無しデータの集合は，パラメータ#1 が α 以下である．従って，このような場合，「パラメータ#1 ＞ α」を追加検査用テストケースの条件として決定する．2 回目以降の検査においては，予め生成していたテストデータ群から条件を満たすテストデータを選択し，入力データとする．ただし，前回以前の検査における入力データと重複するデータは除外する．

以上のように，本手法では，繰り返し未達要素の特定と未達要素を対象装置に実行させることができる入力データの選択を行い，追加検査を実施することで，効率的に未達要素を減らすことができる．

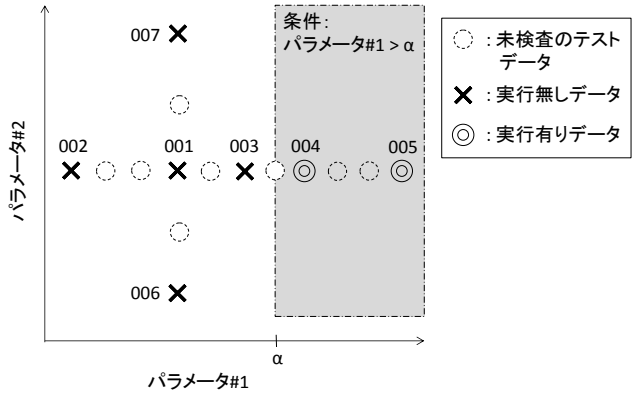

図 5　実行有りデータと実行無しデータとの境界条件の一例を示す図

4. 評価

本章では，提案手法の検査フローに沿って実際にファジング検査を行い，未達要素の特定が行え，追加検査時にその未達要素を実行可能な入力データを選択できているかについて評価する．具体的には，一次検査の結果を用いて提案手法を実施し，二次検査の結果で評価を行う．

4.1 一次検査における準備

検査装置にはファジングツールを，検査対象装置にはネットワークプロトコル A の通信を行う機器を用意する．検査対象装置には予め，GCC（GNU Compiler Collection）のカバレッジ計測オプション付きで検査対象となるソースコードをコンパイルしておく．検査時には，1 つの入力データに対して 1 つの実行履歴を出力し，対応付けられるように運用する．検査装置ではプロトコル A の正常データを読み込ませ，715,997 個のテストデータをランダムに生成した．テストデータにはそれぞれ#0 から#715996 までの識別情報がファジングツールによって振られている．

4.2 未達要素特定

実際に検査を行い，実行履歴から未達要素の特定を行えるかを評価する．

4.2.1 評価条件

本来は，生成したテストデータの中から検査時間や目的に応じて入力データをランダムもしくは任意で選択するのだが，本稿では，10 個の入力データを選択した．選択基準としては正常データである#0 と,プロトコル A のパラメータを任意で 3 種類選び，1 つのパラメータをアノマリとしているテストケースをパラメータごとに 3 個ずつ選択した．図 6 に選択した 1 次検査用入力データを示す.具体的には，識別情報「0000」の入力データはパラメータ#1 に「0x0034」，パラメータ#2 に「0x4000」，パラメータ#3 に「0x0002」といったようにパラメータの値を持っている．そして，識別情報「6500」,「6600」及び「6700」が同じパラメータ#1 にアノマリを持つグループとして分けられる．

入力データの識別情報	パラメータ			
	パラメータ#1	パラメータ#2	パラメータ#3	…
0000	0x0034	0x4000	0x0002	
6500	0x1c34	0x4000	0x0002	
6600	0x8034	0x4000	0x0002	
6700	0xe434	0x4000	0x0002	
7400	0x0034	0x1e00	0x0002	…
7500	0x0034	0x8300	0x0002	
7600	0x0034	0xe700	0x0002	
8100	0x0034	0x4000	0x5002	
8200	0x0034	0x4000	0xb402	
8300	0x0034	0x4000	0x0012	

図 6　一次検査用入力データ（評価時）

検査対象となるソースコードのファイル数は 63 あるが，本稿では，分析する対象をその中からプロトコル A の処理に最も関係するもの 1 つに限定した．

検査基準については,「実行回数が 5 回以上」と設定した．

4.2.2 評価結果

一次検査を実施し，10 個の入力データと 10 個の個別実行履歴を取得した．各入力データと個別実行履歴とを対応付け，総実行履歴を加えた実行情報を図 7 に示す．総実行履歴が検査基準「実行回数が 5 回以上」を満たしていないコードを実行情報から探すと，行番号「87」のコードの実行回数が 2 回で検査基準を満たしていないため，未達要素であることが分かる．図 7 に示してあるコード以外にも検査対象としているコードは存在し，その中にも検査基準を満たさないコードが複数ある．本稿で分析したソースコード 1 つに含まれるコードは 660 行あり（全 1,580 行のうち，コメントや変数定義のような実行に無関係の行は除いている），未達要素は行番号「87」のコードを含めた計 36 個存在した．

	入力データの識別情報	コード行番号				
		87	112	170	216	…
入力データ毎の個別実行履歴	0000	0	3	0	24	…
	6500	0	3	0	19	
	6600	0	3	0	19	
	6700	0	3	0	19	
	7400	0	3	0	19	
	7500	1	3	0	19	
	7600	1	3	0	19	
	8100	0	3	0	19	
	8200	0	3	0	19	
	8300	0	3	0	19	
総実行履歴 ⟶	計	2	30	0	195	…

図 7　一次検査時の実行情報（評価時）

4.3 追加検査用入力データの選択

一次検査で見つかった未達要素を，追加検査時に実行可能な入力データを提案手法で選択できているかを評価する．

4.3.1 評価条件

本来は，未達要素が複数あった場合には，それぞれの未達要素について提案手法を適用し，追加検査用入力データの選択を行う．しかしながら，本稿では，一次検査で見つかった未達要素のうちの 1 つ（行番号「87」のコード）に限定して評価を行う．

4.3.2 評価結果

まず，10 個の入力データを未達要素における実行有りデータと実行無しデータとに分類する．実行させているか否かは図 7 の実行情報の入力データ毎の個別実行履歴部分で判断する．具体的には，識別情報「7500」及び「7600」の入力データが 1 回以上行番号「87」のコードを実行していることが分かるので，実行有りデータとし，そのほかの入力データは 1 回も実行していないので，実行無しデータと分類する．そして，図 6 の一次検査用入力データのパラメータ情報から実行有りデータの集合の属性を表すパラメータの条件を決定する．本稿では条件決定方法として，汎用のデータマイニングツールを利用して実行有りデータと実行無しデータとの境界値を決定した．決定した条件は，「パラメータ#2 の値が“0x6180”以上」である．

次に，二次検査に利用する入力データを選択する．本稿では，既に生成済みの 715,997 個のテストデータの中から，「パラメータ#2 の値が“0x6180”以上」であるという条件を満たし，かつ一次検査で利用していない入力データを 10 個選択する．二次検査で利用する入力データを図 8 に示す．

入力データの識別情報	パラメータ			
	パラメータ#1	パラメータ#2	パラメータ#3	…
7515	0x0034	0x9200	0x0002	…
7525	0x0034	0x9c00	0x0002	
7535	0x0034	0xa600	0x0002	
7545	0x0034	0xb000	0x0002	
7555	0x0034	0xba00	0x0002	
7565	0x0034	0xc400	0x0002	
7575	0x0034	0xce00	0x0002	
7585	0x0034	0xd800	0x0002	
7595	0x0034	0xe200	0x0002	
7605	0x0034	0xec00	0x0002	

図 8　二次検査用入力データ（評価時）

二次検査時の総実行履歴を一次検査時の総実行履歴と合算し，得られた総実行履歴を図 9 に示す．二次検査までの検査によって行番号「87」のコードは総実行回数が 12 回となり，検査基準を満たす結果となった．また，同じテストケース数 10 個で比較しても，行番号「87」のコードにおいて，一次検査では総実行回数 2 回に対して，二次検査では総実行回数 10 回と実行回数が増加していることから，未達要素を集中的に検査できていることが分かる．

	入力データの識別情報	コード行番号				
		87	112	170	216	…
総実行履歴 ⟶	計	12	51	0	313	…

図 9　一次検査及び二次検査合算の総実行履歴（評価時）

5. 考察

5.1 未達要素特定

本稿では，未達要素を特定するために検査密度の指標として実行回数を用いた．パラメータによって実行されるプログラム要素が異なることや，パラメータ値によっても分岐される処理が異なるため，実行回数を用いて検査密度の低い未達要素を特定することができた．しかしながら，一次検査時の個別実行履歴を分析すると，1 つの入力データ

で何回も実行される部分もあれば，正常値でないと実行されない部分，異常値でないと実行されない部分などの違いが総実行回数では区別できないことが分かった．ファジングのように，異常値により発現する脆弱性を見つけることを検査の目的とする場合，異常値でないと実行されない部分が十分な検査密度を満たしているかに着目する必要がある．しかしながら，実行回数を検査密度の指標とした際には，1 つの入力データでしか大量に実行されていない要素がある．また，1 つの検査値でしか実行されていない要素もある．つまり，検査密度としては低いはずだが，実行回数ベースでは回数が多くなり未達要素として検知できていない．ゆえに，実行した入力データ数や，実行させた検査値のバリエーション数を考慮しなければならないことが分かった．さらに，正常値でないと実行されない要素が未達要素として検知される場合があり，本来の目的と合っていない．この場合は，正常値で実行した場合に重みづけを行えばよい．例えば，入力データを 1 つずつ順次に検査対象に送るファジング方法では，1 つの異常なケースに対して応答の有無を確認するために複数回正常なケースを送ることで，正常値で実行した場合に対して重みづけを行える．

また，妥当な検査基準の設定方法にも検討の余地がある．検査基準によって見つかる未達要素の数が異なるため，入力データ数によって検査基準を決めるのか，プログラム要素がとりうる理論上のパラメータ値の数によって決めるのかなど議論が必要と考える．

5.2 追加検査用入力データの選択

一次検査で未達要素とした 36 個の未達要素のうち 1 つの未達要素に関して評価を行った．実行有りデータのパラメータ条件から追加検査用入力データの選択を行い，二次検査では未達要素を実行することができた．推定した実行有りデータと実行無しデータとの境界値については，必ずしも正しいとは言えない．例えば，推定した実行有りデータの条件を満たしていても未達要素を実行できない場合もある．しかし，入力データ数の増加，及び，追加検査を繰り返すことによって境界値周辺のデータをより集められるため，より精度の高い境界値を推定でき，未達要素の実行可能性を高めることができる．

また，本稿では対象としなかった他 35 個の未達要素について一次検査及び二次検査合算の総実行履歴を分析すると，行番号「87」のコードを実行可能な入力データを用いて二次検査を行ったことで，残り 35 個の未達要素のうち 31 個の未達要素が実行され，検査基準を上回っていることが分かった．このため，1 つの未達要素を実行可能な入力データで検査することによって，関連する未達要素も実行できるため，追加検査の効果が予想よりも大きいことが分かった．

6. まとめ

本稿では，検査密度を検査時の同一コード実行回数で定量化し，検査装置のテストケースと検査対象装置の実行履歴とを対応付けることによって，ファジング検査の検査密度を向上させる手法について論じた．このことによって，全組み合わせを網羅した検査を行うなど膨大な検査時間をかけずに，未達要素をピンポイントで追加検査することが可能となった．

今後の課題としては，入力データ数を増やした場合にも同じ効果が得られるか，複数の未達要素が存在する場合にそれぞれの未達要素の関連性を分析することで，未達要素の優先度を決定し，検査効率を上げられるかについては評価できていないことが挙げられる．また，検査密度の観点が 1 つの検査対象要素に対していかに多くのバリエーションの値で検査できているかということに対して，実行回数は 1 つの検査対象要素に対していかに多く実行できているかという点で乖離が生じている点がある．この乖離によって，検査が実行回数ベースの検査基準でクリアできていたとしても，同じ検査値で実行されている場合には検査密度が向上しているとは言えないため，厳密に検査密度を評価する際には実行回数以外の検査指標を検討する必要がある．さらに，検査の十分性を定義することは難しい課題ではあるが，妥当な検査基準を設定する方法についても検討が必要である．

参考文献

[1] “制御システムユーザ企業の実態調査報告書”. https://www.ipa.go.jp/files/000051551.pdf, (参照 2016-07-26).

[2] “サイバーセキュリティ戦略”. http://www.nisc.go.jp/active/kihon/pdf/cs-senryaku-kakugikettei.pdf, (参照 2016-07-26).

[3] “国際的な標準・認証の動向”. http://www.css-center.or.jp/sympo/2015/documents/20150514-22_03kobayashi.pdf, (参照 2016-07-26).

[4] EDSA 100: 2014. ISA Security Compliance Institute – Embedded Device Security Assurance – ISASecure certification scheme.

[5] EDSA 310: 2015. ISA Security Compliance Institute – Embedded Device Security Assurance – Requirements for embedded device robustness testing.

[6] “制御機器認証と拡張について―EDSA 2.0.0(FSA-E/SDLPA/SDA-E)”. http://www.css-center.or.jp/sympo/2015/documents/20150514-22_04a_shimizu.pdf, (参照 2016-07-26).

[7] “制御機器認証と拡張について―EDSA 2.0.0 対応 ERT について”. http://www.css-center.or.jp/sympo/2015/documents/20150514-22_04b_ichikawa.pdf, (参照 2016-07-26).

[8] “ファジング実践資料（テストデータ編）”. http://www.ipa.go.jp/files/000051633.pdf, (参照 2016-07-27).

[9] “Analysis of Mutation and Generation-Based Fuzzing”. http://securityevaluators.com/knowledge/papers/analysisfuzzing.pdf, (参照 2016-08-04).

[10] M. Sutton, A. Greene, and P. Amini. ファジング ブルートフォースによる脆弱性発見手法 ランダムデータの自動注入によるセキュリティテストの実際. 伊藤裕之訳. 毎日コミュニケ

ーションズ. 2008.
[11] Liu, G.H., Wu, G., Tao, Z., Shuai, J.M. and Tang, Z.C.. Vulnerability Analysis for X86 Executables Using Genetic Algorithm and Fuzzing. Third International Conference on Convergence and Hybrid Information Technology. 2008, vol. 2, p. 491-497.