Computer Security Symposium 2019
21 - 24 October 2019

RA: スマートコントラクトの安全性解析にむけたシンボリック実行ツール

知念 祐一郎[1,a)]　矢内　直人[1]　ジェイソン　ポール　クルーズ[1]　岡村　真吾[2]

概要：スマートコントラクトはブロックチェーン上で稼働するプログラムであり，分散型アプリケーションの開発に用いられる．近年の研究ではスマートコントラクトに対し，コントラクトの生成や外部コントラクトの呼び出しといった複雑な動作を通じて不正に送金処理を行うリエントランシー攻撃が報告されているが，既存のスマートコントラクトの検査ツールは単一のコントラクト内部の動作のみを検査対象としており，このような複雑な攻撃を解析できない．本稿では，Ethereum スマートコントラクトを対象に先述した複雑な動作を解析するシンボリック実行ツール RA を提案する．RA はコントラクトの生成や外部呼び出しが起きた状況であっても，プログラムの動作を制御フローグラフを通じて正確に表現できる．これにより，既存ツールに比べコード網羅率を大幅に上昇させるとともに，複雑化している攻撃手法に対する脆弱性を検査することが期待できる．

キーワード：ブロックチェーン, Ethereum, スマートコントラクト, シンボリック実行, 制御フローグラフ

RA: Symbolic Execution Tool Toward Security Analysis of Smart Contracts

Yuichiro Chinen[1,a)]　Naoto Yanai[1]　Jason Paul Cruz[1]　Shingo Okamura[2]

Abstract: Smart contracts are programs that run on blockchains and are used to develop decentralized applications. While attacks against smart contracts involving complex operations such as contract creation and calling external contract have been reported, existing inspection tools only focus on operations within a single contract and thus cannot analyze such attacks. In this paper, we introduce a new symbolic execution tool, named RA, to analyze Ethereum smart contracts including the complex operations described above. RA is able to express a program behavior even against attacks based on the complex operations through generating a control flow graph. Consequently, as well as improving a code coverage in comparison with the existing tools, RA can be potentially used to analyze inspecting attacks based on vulnerabilities to complicated attack methods.

Keywords: Blockchain, Ethereum, Smart Contract, Symbolic Execution, Control Flow Graph

1. はじめに

1.1 研究背景

ブロックチェーンは，不特定多数の参加者からなる P2P ネットワーク上で合意を形成することで一意に決定されるデータ構造である．ブロックチェーン上に保存するデータとして，暗号通貨が持つ通貨残高に加えてプログラムのコードや永続的な変数を格納し，プログラムの実行環境を提供する仕組みをスマートコントラクトという．スマートコントラクトはトランザクションを発行することでプログラムを実行し，そのロジックに従って暗号通貨やスマートコントラクト上で定義したトークン，写真の利用権といったデ

[1] 大阪大学 大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University
[2] 奈良工業高等専門学校
National Institute of Technology, Nara College
[a)] t-yuichito@ist.osaka-u.ac.jp

© 2019 Information Processing Society of Japan

ジタルアセットを送受信する．スマートコントラクトのデファクトスタンダードとなっているのが，Ethereum [1] であり，多数のアプリケーションが開発されている [2][3].

スマートコントラクトの問題) スマートコントラクトの問題点として，ブロックチェーンの透明性によりプログラムが誰でも閲覧可能できる点がある．さらに前項で述べたように金銭的な価値の高いデータを取り扱うという性質もあり，悪意のある第三者に攻撃されるリスクが高い．例えば 2016 年 6 月に起きた The DAO 事件では，リエントランシーと呼ばれる脆弱性を利用し送金を行う関数を不正に再帰呼び出しすることで 60 億円相当の暗号通貨 Ether が盗まれている．また，2017 年のパリティ多重署名ウォレット攻撃では 150 億円相当の Ether が盗まれている[*1]．

このような大規模な被害が頻発する要因として，gas や fallback 関数の存在などスマートコントラクトの機能の特殊性がある．Ethereum スマートコントラクトは Solidity という高級言語を用いて，コントラクトと呼ばれるプログラムの単位として記述する．一方，実行時の挙動および安全性の性質は従来言語によるプログラムとは大きく異なる．これにより，従来言語ではそもそも起こりえないような挙動が開発者の意図しない形で盛り込まれてしまうことが，容易に起こりえる．また，従来言語との違いとして一度デプロイしたスマートコントラクトは変更不可となってしまう点も挙げられる．すなわち，脆弱性を持つコードがデプロイされてしまった場合，長期的に踏み台にされるなど，取り返しがつかないような事態も起こりえる．

スマートコントラクトの安全性解析) スマートコントラクトは高級言語で記述された後，バイトコードに変換されブロックチェーン上にデプロイされる．バイトコードを解析対象とすることには 3.1 で後述するようにいくつかの利点があり，既存研究においても高級言語のソースコードではなくバイトコードを対象とする場合が多い．バイトコードの解析は，コードが仕様を満たすかを述語論理から検証する形式検証 [4], [5], [6], [7], [8] と，コード内の命令からプログラムの制御フローグラフを抽出するシンボリック実行による手法 [9], [10], [11], [12], [13] が代表的である．著者らが知る限り，形式検証による従来研究は Ethereum のバイトコードの抽象化にとどまっており，脆弱性を実際に解析するには至っていない．これに対し，シンボリック実行はプログラムの制御フローグラフを抽出した解析が行える点で，脆弱性が検出しやすいという大きな利点がある．事実としてシンボリック実行による従来研究では，リエントランシーなどの脆弱性を一部検出することに成功している [9], [10], [11]．

しかしながら，既存のスマートコントラクトに対するシンボリック実行ツールでは，コントラクトの中で新しいコントラクトを生成する，あるいは外部にあるコントラクトを呼び出すような状況を解析することができない．一方で，このような動作を伴う攻撃 [14] も指摘されていることから，コントラクトの生成・呼び出しがされる状況であっても正確に解析できることが望ましい．

[*1] https://medium.com/solidified/the-biggest-smart-contract-hacks-in-history-or-how-to-endanger-up-to-us-2-2-billion-d5a72961d15d

1.2 貢献

本稿では Ethereum 上でコントラクト内での新たなコントラクトの生成や，外部コントラクトの呼び出しを解析できるシンボリック実行ツール RA：Reentrancy Analyzer を提案する．RA は新たに生成されるコントラクトの初期化コード・外部呼び出しされたコントラクトと，その呼び出し元となるコントラクトを紐づけた制御フローグラフを生成することで，従来研究では扱えなかったようなプログラムに対するシンボリック実行が可能になる．これにより，解析におけるコード網羅率を大幅に上昇させることが期待できる．

本稿において技術的に解決した課題は，異なるコントラクト同士を紐づけたシンボリック実行の実現にある．詳細は 3.3 節で述べるが，従来のツール [9] では単一のコントラクト内部の動作しかシンボリック実行できなかった．これに対し，本研究ではコントラクト生成においてはその初期化コードを実行し，また外部コントラクト呼び出しに対しては呼び出す関数の関数 ID を用いることで，異なるコントラクト間の動作も解析できるようになった．詳細は 4.1 節に記載する．

本稿では RA についてのコンセプト実証として，コントラクトの生成を行うパターンを用いて実験したところ，Oyente [9] では制御フローグラフが断片化される一方，提案手法では制御フローグラフを正確に表現できることを確認した．

2. 前提知識

本節では本研究を理解するうえで必要となる Ethereum およびプログラム解析の基礎知識について紹介する．

2.1 Ethereum スマートコントラクト

2.1.1 コントラクト

Ethereum では，プログラムをコントラクトとして記述する．これはオブジェクト志向言語におけるクラスのようにブロックチェーン上に格納するデータと，データを処理する関数を持つ．コントラクトは，ブロックチェーン上ではコントラクトアカウントとして存在しており，通常のユーザから暗号通貨を受信したり，プログラムに従い暗号通貨を送信することができる．

2.1.2 EVM

Ethereum は，EVM(Ethereum Virtual Machine) という

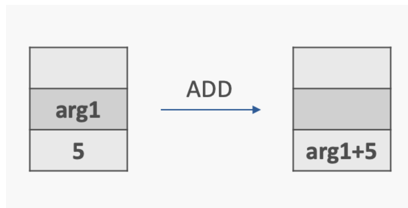

図 1　シンボル変数に対する ADD の様子

プログラムの実行環境を提供している．EVM はスタックマシンで，ほとんどの命令は stack 領域に作用する．スタック領域は 256bit のワードを 1024 ワードまで格納可能で，コントラクトの実行が終了すると消失する．メモリ領域は無限長の配列で，ローカル変数や大きなサイズのデータを保持する為に用いられるが，これもコントラクトの実行が終了すると消失する．ストレージ領域は 256bit のキーで 256bit のバリューを格納するキーバリューストアであり，データの書き込みに多くの gas を消費するが，コントラクトの実行が終了しても消失しない．ストレージ領域や通貨の残高がそのコントラクトアカウントの状態を表すとすると，Ethereum はトランザクションによって状態が遷移するステートマシンと見なせる．

2.1.3　fallback 関数

fallback 関数とは，高級言語 Solidity において外部コントラクトの呼び出しを行った際に指定した関数が存在しない場合や，通貨の送付を行った場合に呼ばれる関数である．高級言語 Vyper では default 関数に相当する．3.2 節で述べるリエントランシー等の脆弱性で利用される．

2.2　プログラム解析

2.2.1　制御フローグラフ

制御フローグラフ (CFG: Control Flow Graph) とは，プログラムの実行の経路をグラフで表現したもので，コンパイラの最適化やプログラムの静的解析で用いられる．分岐せず連続して実行される経路を一纏めにしたものを基本ブロックと呼び，一つのノードに対応する．分岐やジャンプによって到達可能な経路はノードをエッジで結ぶことで表現する．

2.2.2　シンボリック実行

今回解析に用いた手法は，静的解析の一種であるシンボリック実行である．シンボリック実行は，関数の引数といったソースコードからは得られないデータを，シンボル変数という任意の値を表す変数に置き換え擬似的な実行を行う手法である．例えば図 1 のように，シンボル変数 arg1 と整数 5 が stack 領域に積まれている際に EVM バイトコードにおける ADD を実行すると，arg1 と 5 をポップし，arg1+5 という式をプッシュする．

ある経路が実行されるための条件をパス条件という．プログラム開始時のパス条件は恒真で，条件分岐が発生するたびにパス条件に制約が追加される．パス条件にシンボル変数が含まれる場合はその条件式が充足可能か判定することによって，一方の経路のみを実行するのか，あるいは両方の経路を実行するのかを決定する．条件式は一般に一階述語論理式として表現され，充足可能性の判定には SMT ソルバが用いられる．代表的な SMT ソルバとして Z3 がある．

3.　Ethereum スマートコントラクト

本節では本稿で取り扱う Ethereum スマートコントラクトの安全性解析について説明する．まず，安全性解析の対象を述べ，次に主たる問題設定として解析対象とするリエントランシーについて述べる．とくに本稿の対象とするコントラクト生成および外部コントラクトの呼び出しを利用した方法について説明する．

3.1　安全性解析の対象

本稿における解析対象は EVM バイトコードである．これは開発者自身が作成したコントラクトおよび第三者が公開したコントラクトの解析が対象となり，下記のようなメリットがある．

- 高級言語に比べて解析が容易である．
- バイトコードはブロックチェーン上から取得できるため，高級言語のソースコードを入手できないコントラクトの解析ができる．
- 高級言語は仕様の更新頻度が高くまたそのために互換性が保てなくなる場合があるが，バイトコードの仕様は Ethereum ブロックチェーンが更新されない限り変化しない為，ツールの保守性が上がる．

特にコードのみが与えられた状況において，Ethereum クライアント上で実行することなく解析を行うものとする．また，著者らの知る限り EVM バイトコードの難読化はされておらず，本稿においても難読化はされていないものとする．

3.2　リエントランシー

リエントランシーは Ethereum スマートコントラクトにおける代表的な脆弱性で，ある関数の実行中に再び不正にその関数が実行される，即ち再入されてしまう恐れがある．

3.2.1　基本的なリエントランシーの動作原理

fallback 関数を用いた最も基本的なリエントランシーの動作原理について述べる．ソースコード 1 に公開されている[*1]リエントランシーを含むコード例を，ソースコード 2 にリエントランシーに対する攻撃コード例を示す．

コントラクト Fund は key-value ストアである shares を持つ (ソースコード 1-2 行目)．コントラクト Attacker が

[*1] https://solidity.readthedocs.io/en/v0.5.11/security-considerations.html

Fund に対し関数 attack で自身が持つ暗号通貨を送信する (ソースコード 2-4 行目) と，Fund の shares に Attacker のアドレスに対応する送金額が記録される．その後 Attacker が自身の預けた通貨を引き出す (ソースコード 2-5 行目) と，Fund の関数 withdraw で shares に記録された額が Attacker に送信される (ソースコード 1-4 行目)．ここで，Attacker の fallback 関数が実行され，再び Fund の withdraw 関数が呼ばれる (ソースコード 2-8 行目)．Fund は通貨の送信後にそのアカウントから送金された額の記録をリセットする (ソースコード 1-6 行目) が，再び withdraw 関数が呼ばれる時点でこれは実行されていないため，通貨が送信されてしまう．Fund が保有する通貨が全て Attacker に送金されるか，トランザクションが消費する gas 量が制限に達するまでこの再帰呼び出しは継続する．

```
1  contract Fund {
2    mapping(address => uint) shares;
3    function withdraw() public {
4        (bool success,) = msg.sender.call.value(
             shares[msg.sender])("");
5        if (success)
6            shares[msg.sender] = 0;
7    }
8  }
```

ソースコード 1　リエントランシーを含むコードの一部

```
1  contract Attacker {
2    function attack(address target) public
         payable{
3        fund = Fund(target);
4        fund.deposit.value(address(this).balance
             )();
5        fund.withdraw();
6    }
7    function() external payable{
8        fund.withdraw();
9    }
10 }
```

ソースコード 2　リエントランシーに対する攻撃コードの一部

3.2.2　コントラクト生成によるリエントランシー

コントラクトの実行中に新たなコントラクトを生成するのが CREATE 命令である．新たに生成されるコントラクトは，初期化コードとコントラクト本体からなる．初期化コードはコントラクトを生成する際に 1 度だけ実行され，コントラクトがブロックチェーン上に保持するデータの初期化を行う．正常に初期化コードの実行が終了すると元のコントラクトに実行が戻り，コントラクト本体には到達しない．コントラクト本体は，コントラクトアカウントとしてブロックチェーン上に永続的に残る．高級言語でコントラクトを記述しコンパイルすると得られるバイトコードは初期化コードを含んでおり，そのまま解析ツールを適用してもコントラクト本体が解析できない場合がある．

初期化コードは CREATE によって即時に実行されるが，これを fallback 関数のように利用するリエントランシーが Rodler ら [14] によって報告されており，サンプルコードも公開されている*1．

3.2.3　外部コントラクトの呼び出しによるリエントランシー

外部コントラクトに対して通貨を送付したり関数呼び出しを行う際は CALL 命令を用いてトランザクションを発行する．関数呼び出しの場合はトランザクションに送信データを付与する．この内先頭 4 バイトは，呼び出す関数の関数 ID である．関数 ID とは，関数名と引数の型名からなる文字列である関数シグネチャに sha3 ハッシュ関数を適用し，得られたハッシュ値の先頭 4 バイトを取り出したものである．これによって呼び出す関数を指定する．すなわち，外部コントラクトを呼び出す際，バイトコードレベルでは関数名と引数の型しか呼び出し先を特定する情報がなく，意図しないコントラクトに対して呼び出しが実行される可能性がある．また，呼び出されたコントラクトが指定された関数 ID の関数を持たない場合や，通貨の送付のみが目的で送信データが空である場合に実行されるのが 3.2.1 で述べた fallback 関数である．意図しないコントラクトを呼び出してしまった場合，fallback 関数に記述された任意の処理が行われてしまう可能性がある．呼び出し側で消費する gas に制限をかけることでこれを回避できる．

Rodler ら [14] が異なる関数呼び出しの組み合わせに対するリエントランシーを報告しており，サンプルコードも公開されている*1．

3.3　問題点

前節で述べた攻撃方法をシンボリック実行で解析する際，解決すべき技術的課題がある．以下にその具体的な内容について述べる．

まず CREATE で生成されるコントラクトのバイトコードは，シンボリック実行では到達不能である．また，コントラクト生成があった場合に初期化コードを取り出したとしても外部呼び出しとして実行したことにはならない．これにより，初期化コードが fallback 関数のように利用されたコントラクトは正確な解析ができない．また，CALL で外部コントラクトの呼び出しを行うとき，コントラクト同士を独立に解析した場合は呼び出し先を一意に特定できない．特にコントラクトが複数の関数を持つ，即ち再入可能な経路が多い場合，再入する全ての組み合わせをシンボ

*1 https://github.com/uni-due-syssec/eth-reentrancy-attack-patterns

リック実行することが困難となる．

前述した攻撃に対するシンボリック実行を通じた解析に対しては，これらの状況において有効な解析手法を考える必要がある．

4. 提案ツール:RA

本節では新たな解析ツールとして RA (Re-entrancy Analyzer) を提案する．まず，3.3 節で述べた技術的課題を解決するアイデアを述べる．次に，RA の全体像を述べたのち，主たる機能である CFG の拡張と動作原理について述べる．

4.1 アイデア

コントラクト生成に対して，CREATE の引数に着目し生成されるコントラクトのバイトコードを取り出す．そして生成元のコントラクトから生成先のコントラクトの初期化コードに実行を遷移することで，外部呼び出し処理として追跡できるようになる．さらに，これによりコントラクト本体を取得できる為，改めてシンボリック実行を行うことができるようになり，コード網羅率が向上する．

外部コントラクトの呼び出しに対して，トランザクションの送付データとして関数 ID を外部コントラクトに与えることで，どの関数が呼び出されているかを一意に特定できる．このため，コントラクトが再入される場合に取り得る組み合わせを削減した解析が可能となる．

4.2 全体像

RA は，コントラクト生成・外部コントラクト呼び出しを表現できる拡張 CFG を構築する．コントラクトをシンボリック実行する中で CREATE に到達すると新たに生成されるコントラクトの初期化コードに実行が遷移し，CALL に到達すると別のコントラクトに実行が遷移する．それらの中で停止系命令に到達すると元のコントラクトに実行が戻る．新たにコントラクトを生成した場合は，元のコントラクトのシンボリック実行が終了した後で，改めて新しいコントラクトのシンボリック実行を開始する．拡張 CFG では，このコントラクト間を跨ぐ一連の呼び出し関係をグラフとして表現する．

4.3 拡張 CFG の概要

拡張 CFG で CREATE を表現したイメージを図 2 に，CALL を表現したイメージを図 3 に示す．拡張 CFG では，基本ブロックのセパレータとして，JUMP 系命令，停止系命令に加え，CREATE・CALL を利用する．

CREATE・CALL を発行するコントラクトを主たるコントラクト，新たに生成されるコントラクトの初期化コード・外部呼び出しされるコントラクトを従たるコントラクトとする．主たるコントラクトの CREATE・CALL を終端の命令とする基本ブロックから伸びるエッジは，従たるコントラクトの最初の命令を開始の命令とする基本ブロックに到達する．従たるコントラクトの停止系命令を終端の命令とする基本ブロックから伸びるエッジは，主たるコントラクトの CREATE または CALL の次の命令を開始の命令とする基本ブロックに到達する．停止系命令が n 回出現する場合，主たるコントラクトに戻った後の経路も n 本となる．

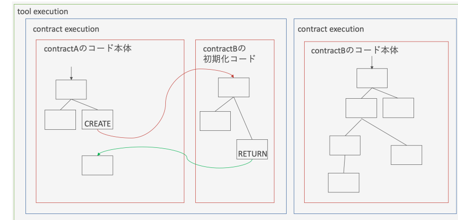

図 2 拡張 CFG でコントラクト生成を表現したイメージ

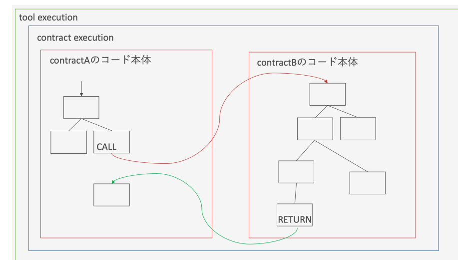

図 3 拡張 CFG でコントラクト呼び出しを表現したイメージ

4.4 動作原理

拡張コントロールグラフのセパレータとなる命令のシンボリック実行の動作原理について述べる．対象は，CREATE，CALL とその亜種である CALLCODE，DELEGATECALL，停止系命令である STOP，RETURN，REVERT である．また EVM は未定義のバイトコードに到達すると停止するが，これが例外処理のために用いられる場合があるため，便宜的に ASSERT とし実行対象とする．CREATE2 はアドレスの生成方法のみが異なる為 CREATE と同一視する．主たるコントラクトへ実行が戻る際に実行環境を復元する為のデータ構造として，`call_stack` を用いる．また，CREATE で生成した新たなコントラクトを格納するデータ構造として，`contract_queue` を用いる．これらによって，再帰的に CALL や CREATE を実行することが可能になる．

4.4.1 CREATE による初期化コードへの遷移

まず現在の実行環境を複製し `call_stack` にプッシュすることで，戻り先の情報を保存する．

CREATE は，stack 領域から 3 つの値 $s0, s1, s2$ を取り出す．$s0$ は，新たに生成されるコントラクトに送られる通貨の額である．$s1$ は memory 領域のオフセット，$s2$ は生成されるコントラクトのデータサイズを示しており，memory

領域の $s1$ 番地から $s1 + s2 - 1$ 番地までのデータを取り出し，新たなコントラクト C_{new} のバイトコードとする．そして stack 領域と memory 領域を維持したまま C_{new} の実行環境を生成し，そのバイトコードの先頭からシンボリック実行を開始する．ここで実行されるのは C_{new} の初期化コードであり，ブロックチェーン上にデプロイされるコントラクト本体には到達しない．初期化コードは，新たに生成されるコントラクト本体のコードを memory 領域にコピーし返り値とする．

4.4.2 CALL による外部コントラクトへの遷移

まず CREATE と同様に戻り先の情報を保存する．外部コントラクトの中で分岐が発生した場合は，call_stack を再帰的に複製しそれぞれの経路で別のオブジェクトとして保持する．これにより外部コントラクトの全ての経路が独立に RETURN を行える．

CALL は，stack 領域から 7 つの値 $si(i = 0...6)$ を取り出す．この内呼び出し先の指定に使われているのが $s1, s3, s4$ である．$s1$ は呼び出す外部コントラクトのアドレスを表す．$s3$ は memory 領域のオフセット，$s4$ は呼び出しの際に送信するデータのサイズを示す．memory 領域の $s3$ 番地から $s3 + s4 - 1$ 番地までのデータを送信データとし，stack 領域と memory 領域を維持したまま外部コントラクトの実行環境を生成する．そしてそのバイトコードの先頭の命令からシンボリック実行を開始する．送信データに含まれる関数 ID はシンボル変数ではなく具体的な値であり，外部コントラクトの実行環境にもこの値が含まれるため，特定の関数を呼び出すことが可能となる．これにより複数の関数を持つコントラクトが複数回呼ばれる場合でも，実行される経路数が爆発的に増えることを回避できる．

CALL 命令の亜種である CALLCODE と DELEGATECALL については外部コントラクトの実行環境を生成する際に維持するデータ領域が異なる．CALLCODE は，stack 領域と memory 領域に加えて storage 領域を維持する．DELEGATECALL は，さらにトランザクションを発行したアカウントを表す msg.sender という領域を維持する．

4.4.3 RETURN による主たるコントラクトへの遷移

RETURN は，返り値を伴いながら実行を正常終了する．stack 領域から 2 つの値 $s0, s1$ をポップし，memory 領域の $s0$ 番地から $s0 + s1 - 1$ 番地までのデータが返り値であることを意味する．

RETURN に到達した際に call_stack が空であれば，シンボリック実行を終了する．空でなければ，ポップした主たるコントラクトの実行状態 S に対して，現在の stack 領域と memory 領域を上書きする．S のプログラムカウンタが指している命令が CREATE であれば，新しいコントラクトの生成を伴う．返り値を contract_queue にプッシュし，そのコントラクトのアドレスを表すシンボル変数を stack 領域にプッシュする．CALL であれば，呼び出しが成功したことを意味する bool 値として 1 を stack 領域にプッシュする．CALLCODE や DELEGATECALL であれば，追加で storage 領域を上書きする．そして S を現在の実行状態としてセットし，プログラムカウンタを 1 つ進めシンボリック実行を再開する．

4.4.4 STOP による主たるコントラクトへの遷移

STOP は，実行を正常終了する．STOP で終了したコントラクトは返り値を持たない．

call_stack が空でなければ主たるコントラクトの実行状態を復元し，さらにプログラムカウンタが指している命令が CREATE である場合は，バイトコードが空であるコントラクトが生成される．この場合は contract_queue に何もプッシュせず，そのコントラクトのアドレスを表すシンボル変数を stack 領域にプッシュする．

4.4.5 REVERT，ASSERT による主たるコントラクトへの遷移

REVERT は実行の巻き戻しを，ASSERT は実行の異常終了を表す．これらで終了したコントラクトは返り値を持たない．call_stack が空でなければポップした主たるコントラクトの実行状態を復元するが，これらに到達するまでに引き起こされたデータ領域に対する作用は全て破棄される為，上書きは行わない．さらにプログラムカウンタが指している命令が CREATE である場合は，0 を stack 領域にプッシュすることでコントラクト生成に失敗したことを表す．

4.4.6 CREATE で生成されたコントラクト本体の実行

コントラクトのシンボリック実行の終了後，contract_queue が空でなければ生成されたコントラクトのバイトコードをポップし，改めてシンボリック実行を行う．

5. 評価

本節ではコンセプト実証を行い提案手法の効果を確認するとともに，提案手法の制限に言及する．

5.1 コンセプト実証

コントラクト生成を含むコントラクトから構成される CFG について既存手法と提案手法を比較することで，提案手法を用いると新たに生成されるコントラクトの初期化コードとコントラクト本体が取り出せることを確認する．

対象となるコントラクト Fund2 のソースコードをソースコード 3 に示す．これは 2 つの関数を持つ．関数 generateNewFund は，リエントランシーを含むコントラクト Fund(ソースコード 1) を新しく生成する．関数 withdraw は，i 番目に生成した Fund から通貨を引き出す．

図 4 は，Fund2 のバイトコードを Oyente[9] に与えて得られた CFG のうち，新たに生成されるコントラクトの初期化コードとコントラクト本体に相当する箇所であるが，

```
contract Fund2 {
    Fund[10] funds;
    function generateNewFund() public{
        funds[funds.length] = new Fund();
    }
    function withdraw(uint i) public {
        int MagicNum = 0xabcdabcdabcd;
        funds[i].withdraw();
    }
}
```

ソースコード 3　コントラクト生成を行うコントラクトの一部

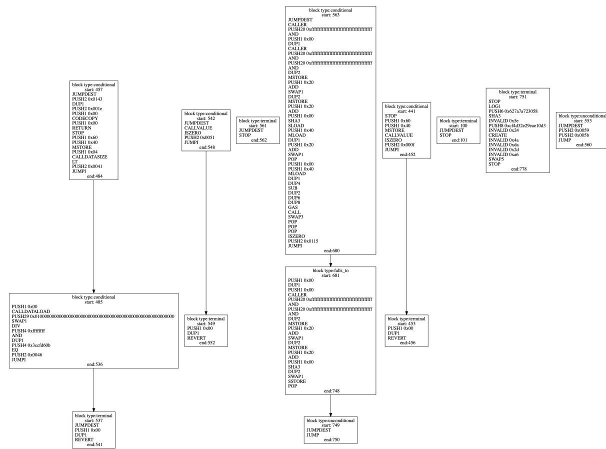

図 4　初期化コードとコントラクト本体の断片化した CFG

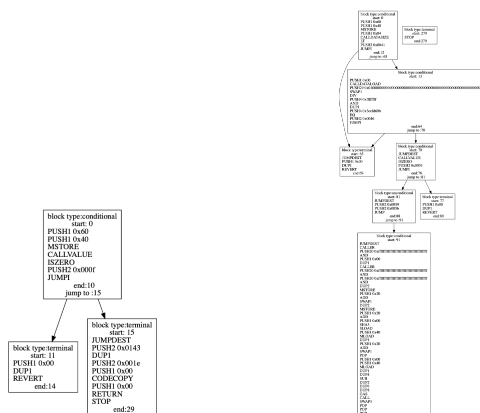

図 5　初期化コードの CFG

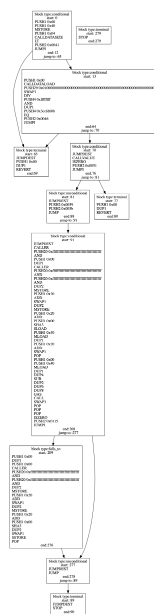

図 6　コントラクト本体の CFG

グラフが断片化していることが分かる．これに対し提案手法を用いて初期化コードを取り出した結果が図 5，コントラクト本体を取り出した結果が図 6 である．Oyente では当該箇所は到達不能であり検査が行われないためコード網羅率が大きく下がるが，提案手法を用いると網羅率を維持できる．

5.2　制限

提案手法の制限は大きく 2 点ある．1 点目は gas の消費量を考慮できないことである．コントラクトが持つ gas 量はシンボル変数として表現しているため取り得る経路の漏れは生じないが，gas の制限を超える実行が行われてしまう可能性がある．2 点目は実行環境を全てシンボル変数として表現していることである．コントラクトが特定量の残高を持つ場合に攻撃を受けるといくらの暗号通貨が奪われるかといったことを解析することは出来ない．

6.　関連研究

本節では関連研究として EVM に対する解析技術としてシンボリック実行，ディスアセンブリ，形式検証の三点に関する研究と，Ethereum の現状調査について紹介する．なお，EVM の解析については文献 [15] でサーベイがされており，興味のある読者はそちらも参照されたい．

シンボリック実行) シンボリック実行による研究は Oyente [9] が先駆けである．Oyente 自体の機能は不完全であるものの, 後続の研究 [10], [11] により多数の拡張がなされている．MAIAN [10] は単一のコントラクトが持つ複数の関数それぞれが呼び出された場合に脆弱性が含まれるかを解析する．Osiris [11] はシンボリック実行に加えて，データの流れを追跡するテイント解析も行うことで，攻撃による制御フローを分析している．また，バイトコードを N-gram 表現する Ether* [12] やガスコストを表現する Gasper [13] も Oyente の拡張である．しかしながら，これらの研究ではコントラクトの生成・外部呼び出しを扱うことができない．

他にもシンボリック実行の研究として，Solidty からバイトコードへのコンパイルとバイトコードからのディスアセンブリを行う Manticore [16], CFG を可視化する Mythrill [17] がある．これらの研究はシンボリック実行の出力を人間にとって読みやすくするという観点の研究である．同様の機能を RA に組み込むことは可能である．

形式検証) EVM の形式検証は Bhargavan ら [4] によって始まった．EVM を形式化した KEVM [5] が代表的成果であるが，安全性検証には至っていない．安全性検証に関する成果としては Securify [6], ZEUS [7] があるが，これらは本稿で取り扱っている外部コントラクトの生成・呼び出しを検出できないことが示されている [14]．Grishchenko ら による形式化 [8] は EVM のみならず安全性の定義にも踏み込んでいる．これと RA のシンボリック実行を併用することで，より精度の高い安全性解析が期待できる．

ディスアセンブリ) EVM の解析に有効な手段の一つがバイトコードのディスアセンブリであり，近年の研究 [18], [19], [20], [21] ではディスアセンブリしたコードの

可読性をさらに上げる内容が主流である．具体的には，ディスアセンブリしたコードから制御フローとスタック情報を表示するE-EVM [18]，疑似コードに変換するErays [19] とPorosity [20]，HTML-like な構文で出力する Vandal [21] がある．これらのディスアセンブリ応用技術を RA に導入することで，より汎用的な解析ができるプラットフォームの構築も期待できる．

仮想通貨の現状) Ethereum に対する攻撃方法の調査として Atzei ら [22] が The DAO 事件や既存の Ethereum 上でのゲームに関する攻撃方法をまとめている．また，最新の結果として Torres ら [23] が Ethereum でどれだけのハニーポットが設けられているか調査している．Torres らが 151,935 個の Ethereum コントラクトを調べたところによると，460 個のハニーポットがあり，実際に攻撃者が 900 万円相当の収入を得ている．

7. 結論

本稿では，コントラクト生成と外部コントラクト呼び出しを制御フローグラフとして表現するシンボリック実行ツール RA を提案した．これによりコントラクトの初期化コードの実行や，複数の関数呼び出しを組み合わせるような動作を伴う攻撃に対する脆弱性を検査できるようになる．今後は，命令毎の gas の消費量と実行中に消費される gas の制限を考慮することで，実行され得ない経路を異常終了できるようにする．また実行の環境変数を任意に与え特定状況下を再現する，あるいは実際のブロックチェーン上から取得できるようにすることで，より柔軟な解析を可能にしたい．

参考文献

[1] Wood, G.: Ethereum: A secure decentralised generalised transaction ledger Byzantium Version, https://ethereum.github.io/yellowpaper/paper.pdf.

[2] Cryptokitties: https://www.cryptokitties.co/.

[3] Cruz, J. P., Kaji, Y. and Naoto, Y.: RBAC-SC: Role-Based Access Control Using Smart Contract, *IEEE Access (Volume: 6)*, IEEE, pp. 12240–12251 (2018).

[4] Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T., Swamy, N. et al.: Formal verification of smart contracts: Short paper, *Proc. of PLAS 2016*, ACM, pp. 91–96 (2016).

[5] Hildenbrandt, E., Saxena, M., Rodrigues, N., Zhu, X., Daian, P., Guth, D., Moore, B., Park, D., Zhang, Y., Stefanescu, A. et al.: KEVM: A complete formal semantics of the ethereum virtual machine, *Proc. of CSF 2018*, IEEE, pp. 204–217 (2018).

[6] Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Buenzli, F. and Vechev, M.: Securify: Practical security analysis of smart contracts, *Proc. of CCS 2018*, ACM, pp. 67–82 (2018).

[7] Kalra, S., Goel, S., Dhawan, M. and Sharma, S.: ZEUS: Analyzing Safety of Smart Contracts., *Proc. of NDSS 2018*, Internet Society (2018).

[8] Grishchenko, I., Maffei, M. and Schneidewind, C.: A semantic framework for the security analysis of ethereum smart contracts, *Proc. of POST 2018*, LNCS, Vol. 10804, Springer, pp. 243–269 (2018).

[9] Luu, L., Chu, D.-H., Olickel, H., Saxena, P. and Hobor, A.: Making smart contracts smarter, *Proc. of CCS 2016*, ACM, pp. 254–269 (2016).

[10] Nikolić, I., Kolluri, A., Sergey, I., Saxena, P. and Hobor, A.: Finding the greedy, prodigal, and suicidal contracts at scale, *Proc. of ACSAC 2018*, ACM, pp. 653–663 (2018).

[11] Torres, C. F., Schütte, J. et al.: Osiris: Hunting for integer bugs in ethereum smart contracts, *Proc. of ACSAC 2018*, ACM, pp. 664–676 (2018).

[12] Liu, H., Liu, C., Zhao, W., Jiang, Y. and Sun, J.: S-gram: towards semantic-aware security auditing for ethereum smart contracts, *Proc. of ASE 2018*, pp. 814–819 (2018).

[13] Chen, T., Li, X., Luo, X. and Zhang, X.: Under-optimized smart contracts devour your money, *Proc. of SANER 2017*, IEEE, pp. 442–446 (2017).

[14] Rodler, M., Li, W., Karame, G. O. and Davi, L.: Sereum: Protecting Existing Smart Contracts Against Re-Entrancy Attacks, *Proc. of NDSS 2019*, Internet Society (2019).

[15] Di Angelo, M. and Salzer, G.: A survey of tools for analyzing ethereum smart contracts, *Proc. of DAPPCON 2019*, IEEE (2019).

[16] Mossberg, M., Manzano, F., Hennenfent, E., Groce, A., Grieco, G., Feist, J., Brunson, T. and Dinaburg, A.: Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts, *arXiv preprint arXiv:1907.03890* (2019).

[17] Mueller, B.: Smashing smart contracts, *9th HITB Security Conference* (2018).

[18] Norvill, R., Pontiveros, B. B. F., State, R. and Cullen, A.: Visual emulation for Ethereum's virtual machine, *Proc of NOMS 2018*, IEEE, pp. 1–4 (2018).

[19] Zhou, Y., Kumar, D., Bakshi, S., Mason, J., Miller, A. and Bailey, M.: Erays: reverse engineering ethereum's opaque smart contracts, *Proc. of Usenix Security 2018*, Usenix Association, pp. 1371–1385 (2018).

[20] Suiche, M.: Porosity: A decompiler for blockchain-based smart contracts bytecode, *DEF con*, Vol. 25, p. 11 (2017).

[21] Brent, L., Jurisevic, A., Kong, M., Liu, E., Gauthier, F., Gramoli, V., Holz, R. and Scholz, B.: Vandal: A scalable security analysis framework for smart contracts, *arXiv preprint arXiv:1809.03981* (2018).

[22] Atzei, N., Bartoletti, M. and Cimoli, T.: A survey of attacks on ethereum smart contracts (sok), *Proc. of POST 2017*, LNCS, Vol. 10204, Springer, pp. 164–186 (2017).

[23] Torres, C. F. and Steichen, M.: The Art of The Scam: Demystifying Honeypots in Ethereum Smart Contracts, *Proc. of Usenix Security 2019*, Usenix Association, pp. 1591–1607 (2019).