Computer Security Symposium 2020
26 - 29 October 2020

内部のバイトコード実行を悪用した スマートコントラクトへの攻撃の早期検知

宮里 俊太郎 [1,a)]　松浦 幹太 [1]

概要：ブロックチェーン上でコントラクトと呼ばれるプログラムを動かすシステムであるスマートコントラクトに対する攻撃として，コントラクトの脆弱性を利用した，Reentrancy 攻撃や Transaction Ordering Dependence 攻撃が現在報告されている．既存研究では，コントラクトの脆弱性を検知する技術が盛んに開発されているが，攻撃に利用されるコントラクトやトランザクションを悪性として検知する研究は少数である．現状の攻撃検知技術の問題点として，検知出来る攻撃の種類が Reentrancy 攻撃だけである事と，トランザクション全てを実行時に監視する事によるオーバーヘッドが挙げられる．本研究では，攻撃の別種である Transaction Ordering Dependence 攻撃も考慮し，Reentrancy 攻撃に関しては，トランザクションのごく一部である，コントラクトを定義するトランザクションのみを静的解析する手法を提案し，それら 2 点の改善を図る．

キーワード：スマートコントラクト，ブロックチェーン, Ethereum，静的解析，攻撃検知

Early detection of attacks which pervert bytecode execution in smart contract

Shuntaro Miyazato[1,a)]　Kanta Matsuura[1]

Abstract: Reentrancy attacks and Transaction Ordering Dependence attacks that exploit vulnerability of contracts are currently reported as a part of attacks against smart contracts, which are systems that run programs called contracts on blockchain. Although technology for detection of the vulnerability has been actively developed, there are few studies that detect contracts and transactions used for the attacks. The problems with the current attack detection technology include the fact that only Reentrancy attacks can be detected and the overhead of monitoring all transactions at execution time. In our research, we consider detection of Transaction Ordering Dependence attack and suggest the method to reduce the overhead of detection of Reentrancy attacks by monitoring only the transactions which define contracts and analyzing them statically.

Keywords: Smart Contract, Blockchain, Ethereum, Static Analysis, Attack Detection

1. はじめに

1.1 研究背景

P2P ネットワークを利用して検証可能な形で取引を記録出来るブロックチェーン技術の誕生により，ビットコイン [1] に代表される暗号通貨など，様々なイノベーションが後押しされている．その中で現在，イーサリアム [2] などのブロックチェーンは，ブロックチェーン上でプログラムを実行する手段を提供しており，このようなシステムはスマートコントラクトと呼ばれる．ビットコインなどの暗号通貨においては，各参加者（ノード）はトランザクションに金額のやり取りを意味する数値のみを書き込むが，スマートコントラクト上では，それに加えてコントラクトというバイトコード実装を記録することが出来る．コントラクトでは例えば，未来に特定の条件を満たせば金銭を受け取る権利を許可するなどの状態遷移のアルゴリズムを，関数や変数を定義して記録することが出来る．加えて，関数の実行が状態遷移の為に行われた場合はその実行も記録される為，コントラクトは締結状況や実行記録が改竄不可能な契約として意味を持つようになる．応用事例としては，電力の使用権，著作権などの移動，あるいは金融産業の自動化が産業界で検討されている．

しかしながら，コントラクトの実装によってはプログラムの脆弱性が含まれる [3] [4] ことが指摘されている．例としては，Reentrancy 脆弱性や Transaction Ordering Dependence 脆弱性といったものが発見されている．また，スマートコントラクトはブロックチェーンの透明性を利用したシステムである為，プログラムは誰でも閲覧可能であり，また価値の高いデータを取り扱うっている為，悪意のある第三者に攻撃される危険性が高い．攻撃者が現に発生した事件も存在する．一例として，かつてドイツの Slock.it 社の「The DAO」プロジェクトで書かれたコントラクトはその中に Reentrancy 脆弱性を含んでおり，それを 2016 年 6 月に悪用されて 5000 万米ドル以上の損失をもたらしたこ

1 東京大学生産技術研究所
Institute of Industrial Science (IIS), the University of Tokyo
a) miyazato@iis.u-tokyo.ac.jp

© 2020 Information Processing Society of Japan

とは有名な事件である [5].

コントラクトプログラムへのそのような攻撃に対する防衛策としては, 多くのセキュリティインシデントと同様, 脆弱性検知と攻撃検知の 2 つの考え方が存在する. 脆弱性検知に関しては, 静的解析 [6] やシンボリック実行 [7] [8] [9] やコンコリック実行 [10] やファジング [11], 形式的検証 [12] [13] [14] [15], 機械学習 [16] といった, 一般的なプログラムのバグ解析に用いられる手法が活発に試されており, 実用に使われているツールも存在する. そのようなツールが想定している使い方は, 被害者となりたくない善良な参加者がローカルでコントラクトを検証してからブロックチェーン上にデプロイするというやり方である. しかしながら, スマートコントラクトの性質上, 一度デプロイされたコントラクトは大幅には変更出来ない上に参加者全員が確認することが出来る. ある時に脆弱性がないと判定されたコントラクトが一度デプロイされてしまった後に, その後の脆弱性検知ツールの進展によって新規の脆弱性が発見されてしまう場合も想定される. よって, 攻撃者が脆弱性検知ツールを悪用する想定も考えられ, 被害者側の自衛策だけでは不十分と言える. このようなことを考慮すると, 攻撃者が攻撃を行う動機を失うようにデプロイ済みのコントラクトを監視する, 攻撃検知技術の必要性が浮かび上がる.

1.2 貢献

本稿では, 脆弱性のある既存コントラクトに対する攻撃を検知する技術について考察し, 既存手法においては, 攻撃者かも知れない相手が提出したバイトコードを直接実行しながら攻撃を検知する方法が主流であり, 手法のオーバーヘッド（メモリ効率性・時間効率性）や早期検知の観点で改善策が考えられることを示す. また, 既存手法で検知出来る攻撃は Reentrancy 攻撃のみであったが, Transaction Ordering Dependence 攻撃も同時に検知出来ることを示す. そして, 出来るだけバイトコードを直接実行せずに, バイトコード記述から実行される関数をトレースすることで攻撃を検知する手法を提案する.

この手法の実証として, 本稿では, ローカルに存在するバイトコードを解析して関数や変数のトレースを行うプログラムを実装し, 先行研究で検知出来るとして示された攻撃のサンプルコード全てを検知出来ることを確認した. しかしながら, 本研究の主要な優位点として想定している, 実際にブロックチェーンネットワークで手法を動かした時のオーバーヘッド削減については, 確認が出来ておらず, 今後公式イーサリアムクライアントの geth [17] を拡張して本手法を導入し, 先行研究と同一条件で大規模な調査を行う必要性がある.

また, スマートコントラクトの仕組み上, この手法では, 中央管理者のような機関ではなく, バイトコードを実行して結果をブロックに書き込むノード（ビットコインで言うマイナーの役割のノード）がトランザクションの中のバイトコードを検証して悪性のコードを受け入れないようなインセンティブが働くのが理想的である. よって, ノードがトランザクションをチェーンに取り込む際に, 同時にバイトコードが攻撃性でないことを確認すると有利になるような, 攻撃検知の動機を与えるブロックチェーンの設計について, 本稿の 6 章で議論する.

2. 前提知識

2.1 スマートコントラクトの原理

現在のスマートコントラクトの中で一番主流であるイーサリアムの仕組みを元に説明する. スマートコントラクトにおいては, ビットコインなどその他のブロックチェーンと同じく, トランザクションのリストとブロックヘッダ（トランザクション全ての集約であるハッシュ値を保持する）を含むブロックをチェーン状に繋げる（親ブロックのハッシュ値を子ブロックに含ませる）ことで状態を管理している. ナンスという任意に変えられる値をブロックヘッダ内に定義し, ヘッダのハッシュ値が一定の値より小さくなるという制約を満たすようなナンス値を不特定多数のノードに探させる（所謂マイニング）ことで, 時間的制約を課し一度保証されたブロックが無効にならないようにしている.

スマートコントラクトがビットコインなどと異なるのは, ステート（各アカウントの預金残高など）を明示的に定義して, そのハッシュ値（厳密にはステートが遷移した部分のみを示すハッシュ値）をブロックヘッダ内に置いていることである. 通常のブロックチェーンでは, 残高はチェーン上に存在せず, 全てのトランザクションの記録の中から未使用の収入(UTXO: unspent transaction output)を探して合計した値としてしか各アカウントの残高は定義されない. スマートコントラクトではさらに, ステートを遷移させる命令としてのバイトコードが定義される. イーサリアムでは Ethereum virtual machine（EVM）上でそのバイトコードが実行されるので EVM バイトコードと呼ばれる. イーサリアムのチェーン上には EVM バイトコードの形でしかプログラムをデプロイ出来ないが, Solidity [18] や Vyper [19] などの, EVM バイトコード出力専門の高級言語が開発されているので, まず高級言語の形でプログラムを書き, それをローカルでコンパイルしてからデプロイするのが通常である.

バイトコードの定義, デプロイおよび実行は以下のようにスマートコントラクト内部で表現される. まず, コントラクト作成者が, コントラクトデプロイの為のトランザクションを生成し, ブロック生成報酬を望むノードがそのトランザクションを含んだブロックを生成することでコントラクトはチェーン上に書き込まれる（デプロイされる）. コントラクト定義用ではない通常のトランザクションは, 金

銭のやり取りやコントラクト上で定義された変数に対する変更を伴う為に from と to（送るアカウントと受け取るアカウント）を定義する必要があるが，コントラクト定義用のトランザクションには to が存在しない（作成者のアカウントしか定義されていない）のが大きな特徴である．バイトコードのハッシュ値もステートの一部であり，それがブロックヘッダに書かれていることで，任意の参加者はあるコードが本当に書き込まれているコードか検証出来る．こうして，定義されたがまだ実行されていないコントラクトは，アドレスと紐つけられて公開される（スマートコントラクトでは，各ノードにアカウント名としてのアドレスがあるのはビットコインなどと同じだが，それに加えてコントラクトにもアドレスが存在し，「コントラクトアドレスにとっての保有残高」もステートとして記録される）．定義されたコントラクト内部の関数は，アクセス権(public などで定義される）を持つ任意のノードによって実行出来る．関数の実行によってステート遷移（金銭や権利のやり取りがある，物の値段が変化するなど）が行われる場合は，必ずその実行はトランザクション生成を伴う．関数を実行する時のトランザクションの to はコントラクトのアドレスとなる．そのトランザクションには関数の計算量を示す gas という値が付け加えられており，トランザクション内部のコードをスタックマシン上で実行してその結果をブロックに書き込む者（マイナー）に報酬として支払われる.斯くして，その正当性が保証されながらコードが実行されステートが変遷する．

今後，コントラクトを定義する為のトランザクションを定義トランザクション，実行する為のトランザクションを実行トランザクションと呼んで区別する．また，マイナーの役割としてブロック生成（ナンス探し）とコード実行の大きく分けて 2 つがあるが，コード実行マイナーと呼んだ時は特に後者の役割に着目している．

2.2 コントラクトが含みうる既知の脆弱性

2.2.1 Reentrancy 脆弱性

Solidity で書かれた脆弱性のあるシンプルなコントラクト（被害者コントラクト）を図 1 に定義し，攻撃者が攻撃の為にまずデプロイするコントラクト（攻撃コントラクト）を図 2 に定義した．図 1 が定義しているシチュエーションは銀行口座のようなもので，正直なノードから被害者コントラクトのアドレスに通貨が送られる（addToBalance を呼び出す）とその値が配列（Solidity では mapping という key-value ストアで表現される）に記録され，送った過去のあるノードは自分が預金した値まで通貨を引き出す（withdrawBalance を呼び出す）ことが出来る．関数 withdrawBalance 内部では，まず関数実行者（msg.sender）の残高に対応する配列を参照して預金が存在しなければ実行は失敗し，存在するならばその値と同額をコントラクトから実行者に送って（call.value）配列の値を 0 にする．この

```
contract VictimBalance {
    // array (address → balance)
    mapping(address => uint256) public userBalances;

    //address myAddress = address(this);
    //private function VictimBalance() {}
    function addToBalance() public payable {
        // user send money to user's account
        userBalances[msg.sender] += msg.value;
    }

    function withdrawBalance() public payable returns (bool) {
        // if user have no money
        if (userBalances[msg.sender] == 0) {
            return false;
        }
        // external function (can be defined by the attacker) call
        // before updating state (vulnerable)
        msg.sender.call.value(userBalances[msg.sender])("");
        userBalances[msg.sender] = 0;
        return true;
    }
```

図 1　シンプルな Reentrancy 脆弱性を持つ被害者コントラクト

```
import "./victim.sol";

contract Attack {
    VictimBalance public victim;

    function attack() public {
        victim.addToBalance{value: 2};
        victim.withdrawBalance();
    }
    fallback() external payable {
        victim.withdrawBalance();
    }
```

図 2　攻撃者が定義するコントラクト

操作は脆弱性を含んでおり，まず，call.value という送金用の関数は外部で（msg.sender 側で）定義を付け加えることが出来る外部呼び出しであり，更に，外部呼び出しを内部の状態遷移（残高を 0 にする）より先に行なっていることが危険である．攻撃コントラクトにおいては，被害者コントラクト のアドレスを指し示すことにより（import victim.sol）図 1 で定義された関数を使うことが出来，また fallback 関数というものを定義することが出来る．この関数は本来，呼び出される関数が見つからなかった場合に予備として実行されてログを出力したりする関数であるが，仕様上，call.value が外部で呼ばれた場合にも必ず実行される．図 2 で fallback 関数の中で更に被害者コントラクトの withdrawBalance が呼ばれていることで，図 2 の attack 関数の中で withdrawBalance を呼び出すと，結果としてその関数が再帰的に複数回呼ばれるという現象が起こる．その間に配列の値は 0 になっていない（外部呼び出しがまだ終了してない）ので，最初の if 文はまだ口座残高が残っているものと解釈する．従って，攻撃コントラクトは，ごく少数の資金を一度被害者コントラクトに送った後で，被害者コントラクトの持つ資金全てを引き出すことが出来てしまう．本来，この脆弱性は操作の順番を変える（外部呼び出しを最後にする）ことで修正出来るし，開発環境によっては call.value 関数は不推奨として注意されるが，これらの注意

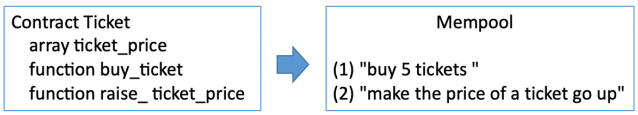


図 3　Transaction Ordering Dependence 脆弱性を持つ
コントラクト の概要図と Mempool

を怠った場合にコンパイルすること自体は出来てしまう．また，あえて脆弱性のあるコントラクトをデプロイした後で正直なノードから多数資金を受け取った後に，外部に攻撃されたと見せかけて資金を奪うやり方があるとすると，被害者コントラクト作成側の自己防衛策はそもそも存在しない．

2.2.2 Transaction Ordering Dependence **脆弱性**

ブロックチェーンとはそもそも，タイムスタンプの信頼出来る機関がない場合に，ブロックを追加した順番でどちらの処理が先に行われていたか保証する仕組みである．よって，トランザクションが作成されて，ブロックに取り込まれるまでの間は，全てのトランザクションはタイムスタンプを持っていない．ブロックに取り込まれてないトランザクションは Mempool という場所に保存されるが，どの順番でトランザクションが作られたかが重要であるにも関わらず分からない場合が存在する．例えば，単純なコントラクトの概念図とそこから生成し得るトランザクションを図 3 に示した．ticketprice というチケットの値段に従って売買が行われ，一部の人間はチケットの値段を上げることも出来るとする．トランザクション（1）の作成者はチケットの購入者として，トランザクション発行直前の値段を見てチケットを買うが，その発行を観察していた攻撃者が（2）のようにチケットの値段を上げるトランザクションを直後に発行する場合がある．この場合，コード実行マイナーがどちらのトランザクションを先に実行するかによって記録が変わってしまう為，（1）の発行者が意図していない高い金額でチケットを買わされる可能性がある．報酬である gas の値を（2）の作成者があえて高く設定する場合のあり，その時は更に（2）が先に実行される可能性が高くなる．攻撃者は，他者のトランザクション発行を観察する必要がある為，攻撃は自動化される場合が想定される．

3.　先行研究

3.1　脆弱性検知ベースの対策

コントラクトの実装者が，実装が満たして欲しい性質を定義した後でそれをコンパイラやツールが証明するという検証方法はいくつか開発されている[12] [13] [14] [15]. 2 章で述べたような脆弱性が存在しないことも，性質の定義を論理や数式で書き下せば証明でき，またいかなる脆弱性にも最終的には対応出来るので拡張性が高い方法ではある．

但し，その定義を具体的に書いて安全性を検証する研究

```
function withdrawBalance() public payable returns (bool) {
    // if user have no money
    if (userBalances[msg.sender] == 0) {
        return false;
    } This variable can change control flow
    // external function (can be defined by the attacker) call
    // before updating state (vulnerable)
    msg.sender.call.value(userBalances[msg.sender])("");
    userBalances[msg.sender] = 0;
    return true;
}
```

tagged

tagged

図 4　動的テイント解析でタグ付けされる例

はまだあまり進んでいない為，コントラクト実装者にとって負担の大きいやり方である．それより自動化を進めて，2 章で述べたような特定の攻撃が成功しないことを確認する方法が現実的なやり方としては主流である．すなわち，コード内で起こりえる様々な状態遷移を出来るだけ辿ってシミュレーションして，攻撃が成立する状態がないか確認するシンボリック実行 [7] [8] のようなやり方である．この方法では，状態の組み合わせが爆発した場合に全てを網羅することが出来ないので，効率的な探索をする為の補助的なやり方として，イーサリアムで公開されているトランザクションの中から典型的な入力パターンを取り出して入力するというコンコリック実行 [10] というやり方もある．いずれにせよ，1 章で述べたように，既にデプロイされたコントラクトに対する攻撃を防ぎ続けるという目的には合致しないという欠点が存在する．また，被害者コントラクトの実装者自体は損をしない形の攻撃であると，攻撃者があえて脆弱性のあるコントラクトを用意する方法も考えられる．被害者コントラクトの実装者以外によっても実行出来る，攻撃自体を見つけられる方法が必要とされる．

3.2　攻撃検知ベースの対策

コントラクトの脆弱性検知の研究に比べて，攻撃検知の研究は比較的少ない. Sereum [20] というツールがこの戦略の代表例である．ただし，Reentracny 攻撃のみに限って分析していることに注意が必要である．この研究では，実行トランザクションに着目し，動的テイント解析という手法を検知の為に使っている．図 2 で言う所の，attack 関数を実行するトランザクションを攻撃者が作成した後に，監視人を置いて検知するというシチュエーションを想定している（例えば，コントラクト実装者ではなく，スマートコントラクトシステム全体に責任を負う開発者が定期的に，発生したトランザクション全てを監視する）．動的テイント解析とは，プログラムの実行と同時に，ある特定のデータに対して属性情報（テイントタグ）を設定して（図 4）シャドウメモリという領域に保存することで，データのフローを解析する技術である．一般的に，脆弱性検知や個人情報流出検知などに使われる．2 章の例だと，まず，攻撃側が呼び出している関数を辿って被害者コントラクトに行き着いた所で，図 4 のように変数にタグを付ける．Reentrancy 攻撃が成功する理由は，withdrawBalance 関数の中で，if 文による制

御フローの管理が無効化したまま送金が複数回行われることである（if 文が存在しない場合は元々送金を制限するつもりがないので不正な送金という概念が存在しない）．よって，if 文があった場合はその結果に影響する変数（ここでは userBalances[msg.sender]）にテイントタグが付けられる．テイントタグは変数の依存関係を出来るだけ追跡するように付けられるので，もし userBalances に影響する変数があったらそれにもタグが付けられる．タグのついた変数は，外部呼び出し call.value から抜け出すまではロックされる．withdrawBalance の再帰的呼び出しは，if 文を回避したとしても，被害者コントラクトの残高がなくなった時点で強制的に失敗し，最後に呼び出された call.value 内では userBalances[msg.sender]に 0 を書き込む命令が実行される．その時にはロックされた変数に書き込みが行われていることになるので，違反として検知される仕組みである．この手法は攻撃検知である為に 1 章で述べたような，脆弱性検知にはない利点を持っている．そして更なるこの手法の利点を示す為に，著者は既存の脆弱性検知ツールの弱点を提示している．攻撃者は Reentrancy 攻撃の難読化（ここでは検知ツールに発見されにくいように攻撃を複雑にさせるという意味で使っている）を行う場合があると筆者は指摘し，例として下記の 3 つを挙げている．

- Cross-function Reentrancy（複数の別の関数を経由した上でなら送金する関数を複数回呼び出せる脆弱性）
- Delegated Reentrancy（ライブラリとして公開されているコントラクトを被害者コントラクトが使っていた場合に，ライブラリコントラクトと被害者コントラクトの組み合わせで初めて生じる脆弱性）
- Create-based Reentrancy（被害者コントラクトの内部で更にコントラクトを新しく生成する命令があった場合に，コントラクト作成時に呼ばれる初期化関数が攻撃者の定義出来る関数を呼んでいることで生じる脆弱性）

具体的なコードとして，著者は図 4 と図 5 のようなコントラクトを示している．図 4 は Cross-function Reentrancy 脆弱性を持ったコントラクト を示している．etherBalance 変数は call.value 直前に変更される為，withdrawAll 関数を直接複数回呼び出す攻撃は出来ないが，tokenBalance 変数は call.value 直後に変更されるので，call.value 関数の中で transfer 関数を呼び出して不正な送金をすることは出来る．図 5 は Delegated Reentrancy 脆弱性を持ったコントラクトを示している．Victim コントラクトの中の delagetecall 命令によってライブラリの Lib コントラクトが呼び出されており，Victim コントラクトだけを見ても Lib コントラクトだけを見ても，Reentrancy 攻撃が成立するかどうかは分からない．加えて検知を困難にしているのは，ライブラリは更新されることがあり，Victim コントラクトをデプロイする

```
1  mapping (address => uint) tokenBalance;
2  mapping (address => uint) etherBalance;
3
4  function withdrawAll() public {
5    uint etherAmount = etherBalance[msg.sender];
6    uint tokenAmount = tokenBalance[msg.sender];
7    if (etherAmount > 0 && tokenAmount > 0) {
8      uint e = etherAmount + (tokenAmount * currentRate);
9      etherBalance[msg.sender] = 0;
10     // cannot re-enter withdrawAll()
11     // However, can re-enter transfer()
12     msg.sender.call.value(e)();
13     // state update causing inconsistent state
14     tokenBalance[msg.sender] = 0;
15   }
16 }
17 function transfer(address to, uint amount) public {
18   // uses inconsistent tokenBalance (>0) when re-entered
19   if (tokenBalance[msg.sender] >= amount) {
20     tokenBalance[to] += amount;
21     tokenBalance[msg.sender] -= amount;
22   }
23 }
```

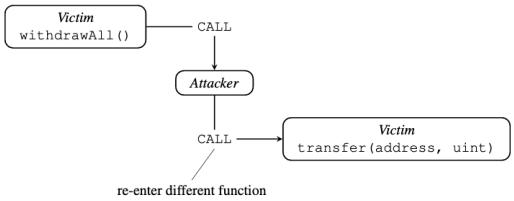


図 4　Cross-function Reentrancy 脆弱性 [19]

```
1  library Lib {  // Library contract
2    function send(address to, uint256 amount) public {
3      to.call.value(amount)(); // CALL
4    }
5    // ...
6  }
7  contract Victim {
8    mapping (address => uint) public credit;
9    Lib lib; // address of library contract
10   // ...
11   function withdraw(uint amount) public {
12     if (credit[msg.sender] >= amount) {
13       // DELEGATECALL into Library
14       address(lib).delegatecall(
15         abi.encodeWithSignature("send(address,uint256)",
16                                 to, amount));
17       // state update after DELEGATECALL
18       credit[msg.sender] -= amount;
19     }
20   }
21 // ...
```

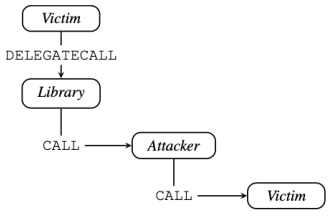


図 5　Delegated Reentrancy 脆弱性 [19]

瞬間には脆弱性があるかどうか完全には分からないという事実である．基本的にコントラクトそのものは更新出来ないが，コントラクト内で別のコントラクトを呼び出す場合にはコントラクトアドレスで指示する為，アドレスを変数とすることで，呼び出すコントラクトを変えて実質的にライブラリの実装を変えることが出来る．従って，被害者側

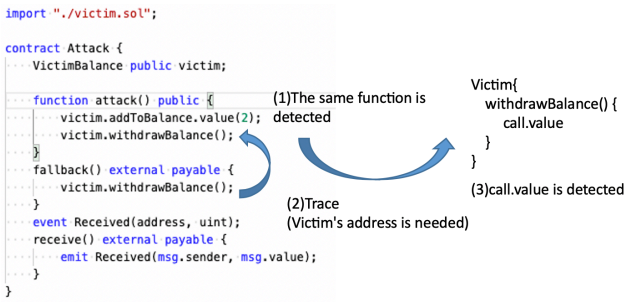


図 6 トレース による Simple Reentrancy 攻撃の検知

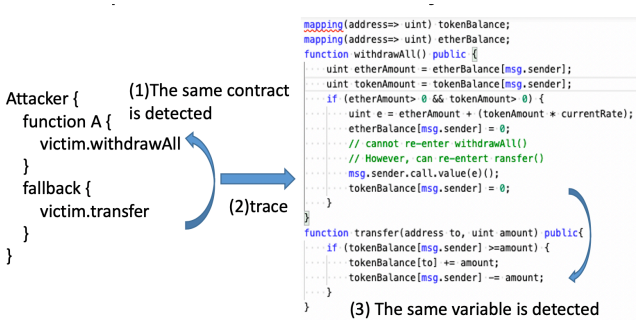


図 7 トレース による
Cross-function Reentrancy 攻撃の検知

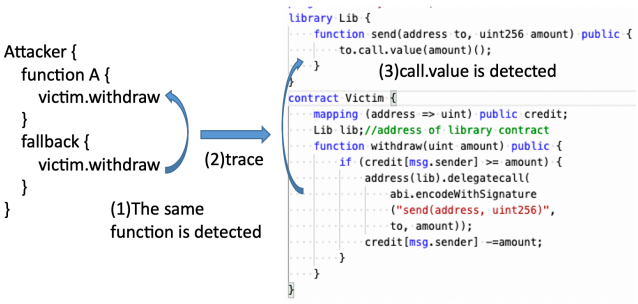


図 8 トレース による
Delegated Reentrancy 攻撃の検知

がローカルで行う解析には限界があり，攻撃者が条件を整えて実際に行う攻撃を見るランタイム検知でしか本当に脆弱かどうかは分からないというのが著者の主張である．攻撃検知の後にどういう対策を行えば良いのか著者は明確には述べていないが，おそらくは，トランザクションがブロックに取り込まれるまでに除外する，トランザクションを発行したアドレスを凍結するなどの対策が考えられる．

4. 提案手法

既存手法 Sereum [20] の問題点としては，Reentrancy 攻撃のみの検知であることに加えて，攻撃検知という恩恵の割にはオーバーヘッドが大きいということが挙げられる．実行トランザクションの実行に用いる通常の時間の 9.6% の時間，通常のメモリ消費の 5.5% がオーバーヘッドとなる．筆者らは，そもそもブロック生成の時間に比べてトランザクションのコード実行に使われる時間が短いので，平均するとブロック生成時間の 0.034% ほどの時間オーバーヘッドで済むとは述べている．しかしながら，良く知られているように，ブロックチェーンは世界規模でやると電力や計算量を莫大に消費する為，小規模なオーバーヘッドの改善でも実質的にもたらされる節約量は大きい．更に付け加えられると，コントラクトへの攻撃は重大なインシデントを引き起こすが，実際に起こっている数は少ないという分析結果もある [4] ため，トランザクションのほとんどを占める正常なコードに対して計算資源を使うのはそもそも避けるべきであるとも考えられる．そこで，本研究では，実行トランザクションの全てを見るのではなく，出来るだけ監視対象のトランザクションを絞って検知ツールを動かすことで計算コストを下げて，かつ早期に攻撃を発見することが出来る方法を提案する.Reentrancy 攻撃に関しては，コントラクトを実行するトランザクションを無視し，コントラクトを定義するトランザクションのみを 監視する．また，新しく Transaction Ordering Dependence 攻撃も検知出来る手法を提案するが，構造上この攻撃は，コントラクト内の関数を実行するトランザクション の段階でしか検知することが出来ない．

4.1 Reentrancy 攻撃の検知

Reentrancy 攻撃に関しては，必ず攻撃の前に攻撃者は悪性のコントラクト（図 2 参照）をデプロイしなければいけない．既存手法では，このコントラクトが実行トランザクションに呼び出されるまでは無視していたが，これを定義トランザクションの時点で（デプロイ時に）コード実行マイナーが発見する早期検知を行うことで監視対象のトランザクションを絞る．スマートコントラクトの構造上，定義トランザクションは実行トランザクションより圧倒的に少ない為，監視対象が減る．具体的な原理としては，コントラクトのデプロイ直前に，例えば図 6，図 7，図 8 のように関数や変数の追跡を行う（実際は Solidity コードではなく EVM バイトコードを解釈する）．なお，イーサリアム内部のバイトコードにおいて，関数は Method ID（メソッド名と引数の型をスペース無しでまとめた文字列をハッシュ化し先頭 4 バイトを取り出したもの）という数値で管理され，変数はコントラクト内部で定義された順番の数値（0x00 など）で区別・管理される．

図 6 ではシンプルな Reentrancy 攻撃を検知している．まず，Reentrancy 攻撃に必須の fallback 関数の中で，外部関数が呼び出されている場合にのみ，そのコントラクト内の別の手続きで同じ外部関数が呼び出されているか確認する．同じ外部関数を発見したら， その元であるコントラクトまでトレース を行い(ブロックチェーンネットワーク上からコントラクトを取り出すことを想定している)，呼び出される関数の実装を発見する．そして，その関数の中で ，送金用の call.value 関数が使われていた場合，必然的に同じ関数が不正に複数回呼び出されると分析される為，攻撃とみなす．攻撃用コントラクトと認定された場合，コード実

行マイナーはそのコントラクトをデプロイせずに破棄する．

図 7 では Coss-function Reentrancy 攻撃を検知している．シンプルな Reentrancy 攻撃とは違い, fallback 関数内部で呼び出される外部関数と，攻撃用の関数（function A）で呼び出される関数が異なる．よって，関数の完全一致ではなく，同じコントラクトに由来する関数を発見した時点でトレース を開始する．トレースされた 2 つの関数のうち，攻撃用関数からトレース された関数中で送金用の call.value が呼ばれていた場合, call.value の後に変更されている変数を全て記録する（ここでは tokenBalance[msg.sender]）．その変数が, fallback 関数からトレース された関数内で使用されている場合, call.value によって被害者コントラクト側が意図しない変化が起こると分析し，攻撃とみなす．

図 9 では Delagated-Reentrancy 攻撃を検知している．シンプルな Reentrancy 攻撃と同様に，同じ外部関数が fallback 関数内と別の場所の 2 箇所で呼び出されるのを発見する．その後外部関数を追跡し，その外部関数の中でさらに delegatecall 命令で別の関数が呼び出されている場合はそれもまた追跡する．追跡の結果，最終的に送金用の call.value が発見されたら，攻撃とみなす．

Create-based 攻撃に関しては，図は複雑になる為省略するが，攻撃者は，被害者コントラクトを継承した上で，関数をオーバーライドで書き換えることで, fallback 関数による呼び出しのように送金用の関数を複数回呼んでいる．検知する為には，まず，外部コントラクトの関数に対してオーバーライドを行っている関数を攻撃コントラクトの中に発見し，その中で更に外部関数が呼ばれていた場合追跡する．外部関数の中でコントラクト作成時を行っていて，さらにコントラクト作成時のコンストラクタ関数の中で，オーバーライドされた先程の関数が発見された時，関数呼び出しのループが発見されることになる為，攻撃とみなされる．

これらの手法の為には，攻撃コントラクトデプロイ時に被害者コントラクトが明確に分かる必要があるが，攻撃者が被害者コントラクトアドレスを変数にして，デプロイ時とは違う数へ更新することで攻撃を成立させる場合もある．この場合も網羅する為に，「別のコントラクトアドレスを示す変数」を変える操作に限って実行トランザクションを監視するやり方も拡張として考えられる．

4.2 Transaction Ordering Dependence 攻撃の検知

Transaction ordering dependence 攻撃に関しては，実行トランザクションの順番に関する攻撃なので，定義トランザクションの時点で発見するのは困難であると妥協する．図 9 にあるように, Mempool を常に監視し，同じコントラクトの同じ変数が参照されている場合，順番によって結果が異なる場合には適切ではないとする．参照だけで値を変化させる訳ではない場合は安全なので，変数の変更命令が実行トランザクション内部であった場合にのみ Mempool 内の他のトランザクションを確認する．参照と変数書き込み両方の命令が共存した場合には，書き込み用のトランザクション のみを Mempool から削除し，キャンセルする．

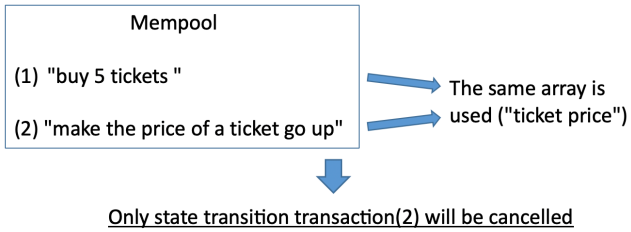


図 9 トレース による
Transaction Ordering Dependence 攻撃の検知

5. 実装と評価の進展

4 章で提案した解析法は，最終的には，マイナーによって使われるようにイーサリアム 公式クライアント（Go 言語で書かれた geth など）の上に実装され，先行研究と比較する必要がある（先行研究は geth 上に実装されている）．しかし，現在は，ローカルに攻撃者側のバイトコードと被害者側のバイトコードが存在する状態で検知する為の，アルゴリズム確認用のコードを Python で実装した所である．このコードにおいて，まずは Reentrancy 検知の網羅性の確認として， Sereum [20]が検知出来るとして論文に紹介されていた，攻撃者サンプルコード [21] を使用した．このサンプルコードには，シンプルな Reentrancy から, 3 種類の難読化された Reentrancy までを含んでいるが，全て検知出来ることが確認された．また，Transaction Ordering Dependence 脆弱性に関しても，被害者コントラクト があった場合にどの関数とどの関数を呼び出されれば成立するか判定するコードをかいて，自作のサンプルバイトコードで検証した．

今後のタスクとしては, geth 上にこのアルゴリズムを実装し，大量のトランザクション を検証させて ，オーバーヘッドの精密な確認と，偽陽性の小ささを確認する必要がある．

6. 検知手法が使われるようになる為の議論

既存研究 [20] は中央管理者が行う検知も想定しているが，悪性のトランザクションをマイニング直前に排除する為には，出来るだけコード実行マイナーがついでに行うような攻撃検知が望ましい．しかしながら，当然検知ツールは計算量を使う為に，コード実行マイナーにどうやって検知させるインセンティブを与えるかが重要となる．参考になる方法として, Proof of Stake プロトコルにて用いられる Slashing アルゴリズム [22] がある．これは，あるブロックの子として成立する 2 つ以上のブロックを作って署名するなど，チェーンの枝分かれ（収束の不安定化）に繋がると考えられるマイニングを行ったマイナーに罰則を与えるア

ルゴリズムである．マイニングと同時にマイナーは自分の持っている資産の一部をデポジットとして差し出し，不正が発覚すると資産の一部が無効になるというプロトコルが存在する．これと同じように，実行したコードが不正でないことを示す署名をコード実行マイナーに提出させ，その後コードに不正が発覚したということが合意されたら署名者の資産の一部が没収されるといったプロトコルが考えられる．資産の没収を避ける為に，コード実行マイナーは検知ツールを常に動かした方が得となる．この仕組みを取ると，最終的に不正なコードを含むトランザクションを誰が発見して告発し，最終的にどう合意するかという問題も考えられる．一つには，少数の代表者・責任者が議論を行い，全員が署名を行うことで関わるコントラクト，トランザクションを全て無効にするというやり方が考えられる．議論をオンラインの公開の場（イーサリアムにおいてコントラクトが公開されている Etherscan [23] など）で行うことで透明性を担保し，あくまでコード実行マイナーが攻撃検知を怠った場合の罰則として実施することで，実際にこれが執行される可能性を低く抑えることも検討される．

7. 終わりに

本稿では，スマートコントラクト上ではコントラクト定義の脆弱性を悪用した攻撃がいくつか確認されており，ローカルで被害者側が行う脆弱性検知だけでは防げない場合があるということを紹介した．攻撃検知に着目とした手法は，実行トランザクション全てを監視する手法が代表的であり，オーバーヘッドの観点で効率化が望めることを示した．更にその具体策としての手法を提案し，現時点の実装状況と，今後のタスク，現実にスマートコントラクトプロトコルの中でその手法がどう使われるのが適正であるかを議論した．今後の展望としては，残存している実装の課題点を埋めて評価し，安全にコントラクトを保護するシステムの実現を目指す．

参考文献

[1] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. http://www.bitcoin.org/bitcoin.pdf, 2009.

[2] Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform. https://github.com/ethereum/wiki/wiki/White-Paper, 2014.

[3] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In Principles of Security and Trust, pages 164--186, 2017.

[4] Daniel Perez and Benjamin Livshits. Smart contract vulnerabilities: Does anyone care? CoRR,abs/1902.06710, 2019.

[5] Understanding the dao attack. https://www.coindesk.com/understanding-dao-hack-journalists, 2016.

[6] Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bunzli, and Martin T. Vechev. Securify: Practical security analysis of smart contracts. In Conference on Computer and Communications Security, pages 67--82, 2018.

[7] Petar Tsankov Dana Drachsler-Cohen Anton Permenev, Dimitar Dimitrov and Martin Vechev. Verx: Safety verifcation of smart contracts. In IEEE Symposium on Security and Privacy, pages 414--430, 2020.

[8] Bernhard Mueller. Smashing smart contracts.9th HITB Security Conference, 2018.

[9] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In Conference on Computer and Communications Security, pages 254--269, 2016.

[10] Konrad Weiss and Julian Schtte. Annotary: A concolic execution system for developing secure smart contracts. In Computer Security ESORICS, pages 747--766, 2019

[11] Bo Jiang, Ye Liu, and W. K. Chan. Contract-fuzzer: fuzzing smart contracts for vulnerability detection. In ACM/IEEE International Conference on Automated Software Engineering, pages 259--269, 2018.

[12] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. A semantic framework for the security analysis of ethereum smart contracts. In Principles of Security and Trust, pages 243--269, 2018.

[13] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. Online detection of effectively callback free objects with applications to smart contracts, 2017.

[14] E. Hildenbrandt, M. Saxena, N. Rodrigues ,X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu, and G. Rosu. Kevm: A complete formal semantics of the ethereum virtual machine. In IEEE Computer Security Foundations Symposium, pages 204--217, 2018.

[15] Zheng Yang, Hang Lei, and Weizhong Qian. A hybrid formal verifcation system in coq for ensuring the reliability and security of ethereum-based service smart contracts. IEEE Access, pages 21411--21436, 2020.

[16] P. Momeni, Y. Wang, and R. Samavi. Machine learning model for smart contracts security analysis. In International Conference on Privacy, Security and Trust, pages 1--6, 2019.

[17] Go Ethereum, https://github.com/ethereum/go-ethereum, 2020.

[18] Solidity language documentation. https://solidity.readthedocs.io/en/v0.6.7/, 2020

[19] Vyper―vyper documentation. https://vyper.readthedocs.io/en/latest/, 2020

[20] Michael Rodler, Wenting Li, Ghassan O.Karame, and Lucas Davi. Sereum: Protecting existing smart contracts against re-entrancy attacks. In Network and Distributed System Security Symposium, 2019.

[21] Re-Entrancy Attack Patterns. https://github.com/uni-due-syssec/eth-reentrancy-attack-patterns, 2020.

[22] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. CoRR, abs/1710.09437,2017.

[23] Etherscan―ethereum (eth) blockchain ex-plorer. http://etherscan.io