Ethereum Smart Contract でのコンパイラによる脆弱性への影響調査

加道 ちひろ[1] 矢内 直人[1] ジェイソン ポール クルーズ[1] 今村 光良[2] 岡村 真吾[3]

概要：Ethereum スマートコントラクトにおいて, 脆弱性は重要な問題である. 脆弱性の検知ツールの開発は様々されているが, 脆弱なコードを除外するコンパイラそのもの有効性については著者の知る限り未調査である. 本稿では, Ethereum スマートコントラクトにおいて脆弱性があるプログラムがどれだけ作成されているか, コンパイラのバージョンごとに調査する. 具体的には, まず 2022 年 4 月までに Ethereum ブロックチェーンに保存されているコントラクトのうち, ソースコードが公開されているコントラクト 626,171 件を収集した. それらのコントラクトに対して, 既存の脆弱性解析ツール SmartCheck を使用することで, 各コントラクトの脆弱性を解析した. 解析結果として, 重要度の高い脆弱性として知られる Unchecked Call, Locked Money, Using tx.origin の 3 つに注目したところ, コンパイラのバージョン更新に伴い, 脆弱性の発生率が減少傾向にあることを確認した. また, コンパイラ更新の弊害として, 更新直後に一時的に脆弱性の発生数が増加するが, 半年程で安定することを確認した. なお, Unchecked Call 脆弱性は減少傾向にあるもののまだ発生率が高く, 今後の対策が必要である.

An Empirical Analysis on Impact of Compilers on Vulnerabilities in Ethereum Smart Contracts

Chihiro Kado[1] Naoto Yanai[1] Jason Paul Cruz[1] Mitsuyoshi Imamura[2] Shingo Okamura[3]

1. 序論

1.1 はじめに

ブロックチェーンは, 近年では暗号通貨による金銭取引だけではなく, 分散アプリケーションのプラットフォームであるスマートコントラクトとしても利用されている. ここでいうスマートコントラクトとは, 暗号通貨の取引情報に加えプログラムのコードをブロックチェーン上に保存することで, 自動的に取引を行うことができるシステムである. ブロックチェーン上に実装可能なスマートコントラクトのプラットフォームとしては, Ethereum [1] が最も一般的なスマートコントラクトとして知られている. （本稿では以降, Ethereum スマートコントラクトを単にスマートコントラクトと呼称する.）

スマートコントラクトはブロックチェーンの性質に起因して, セキュリティに関する様々な問題が存在する [2]. まず, ブロックチェーン上にプログラムコードが保存されることから, 誰でも情報を閲覧できるブロックチェーンの透明性に起因して, 攻撃者にとってもプログラムの解析が容易となる. これにより, 脆弱性をついた攻撃を誘引しやすい [3]. 次に, スマートコントラクトではプログラムの実行に手数料が必要であり, また, そのプログラム自体も金銭的な価値のあるものを扱うことが多く, 攻撃が金銭的な被害に直結する. 例えば, 2017 年にはパリティウォレット攻撃によって約 1 億 5000 万 US ドルの被害が発生している[*1].

上述した背景から, スマートコントラクトの脆弱性に対して, それらを解析するツールの研究開発 [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14] や, コンパイラの更新を通

[1] 大阪大学
Osaka University
[2] 野村アセットマネジメント
Nomura Asset Management Co., Ltd,
[3] 奈良工業高等専門学校
National Institute of Technology, Nara College

[*1] https://techcrunch.com/2017/11/07/a-major-vulnerability-has-frozen-hundred-of-millions-of-dollars-of-ethereum/

じて脆弱なコードがコンパイルできないような対策機能の追加が実施されている[*2].

既存の脆弱性解析ツールの評価 [15], [16], [17], [18], [19], [20], [21] は様々に行われている一方, コンパイラの改善によって実際のスマートコントラクトの脆弱性がどのように変化してきたか, 著者の知る限り調査されてこなかった. これは, 上述した通りコンパイラはスマートコントラクトの脆弱性対策として重要な機能を持つにもかかわらず, その有効性が明らかにされていないことを意味する. このため, スマートコントラクトの脆弱性に対し, 有意義な解析ツールの研究開発を行う指針となるような, コンパイラの影響含めた脆弱性の現状調査が望まれる.

以上を踏まえ, 本稿の主たる問いは以下のとおりである.

- Q1: スマートコントラクトの脆弱性はどのように変化しているか.
- Q2: コンパイラのバージョンが各脆弱性にどう影響している.

上述した問いは非自明である（詳細は 2.3 節を参照されたい）. 本稿では重要度の高い脆弱性に着目して, これらの問いに関する検討を行う.

1.2 貢献

本稿では, Ethereum スマートコントラクトの脆弱性がコンパイラの更新とともにどのように変化しているか, 公開されているプログラムから調査する. 具体的には, 2015 年 7 月 31 日から 2022 年 4 月 30 日までに作成されたコントラクト 626,171 件に対し, コンパイラのバージョンごとにコントラクトを分類し, それらのコントラクトが持つ脆弱性を既存の脆弱性解析ツール [5] を通じて確認した. それによって, コンパイラによる脆弱性対策が有効であるか, それとも今後のさらなる対策が必要であるか議論している. 結果として, コンパイラの更新が脆弱性の対策として有効であることを部分的に明らかにしている.

具体的には, とくに重要度 [5] が最も高い脆弱性に着目して検討したところ, コンパイラの更新に伴い脆弱性の件数が削減されていることを確認した. また, 各コンパイラは更新直後に一時的に脆弱性が増える傾向にあること, すなわち脆弱性が実際に減少する安定時期まで数か月程度時間を要することも確認している. このコンパイラの更新直後の不安定さはスマートコントラクトによるコードの開発者の懸念 [22] でもあるが, 本稿では半年後にはおおむね安定すること, すなわち, 更新したコンパイラを使うほうが望ましいことを示している. なお, 重要度の高い脆弱性として Uncheck Call のみがコンパイラによる対策は十分ではなく, 今後の研究開発による対策が望ましい.

[*2] https://github.com/ethereum/solidity/releases

2. 技術的背景と本研究の課題

本節ではまず技術的背景として Ethereum スマートコントラクトの概念及び関連研究について紹介する. 次に, それらを踏まえた本研究の課題について述べる.

2.1 Ethereum スマートコントラクト

Ethereum において, スマートコントラクトはブロックチェーン上にデプロイされ, 「Ethereum Virtual Machine (EVM) の文脈で決定的に実行される変更不可能なプログラム」として扱われる. これはブロックチェーンの改ざん不可能性を通じてスマートコントラクトコードの信頼性が保証されること, すなわち, 一度デプロイされたコードは変更も削除もできないことを意味する. また, スマートコントラクトの関数は誰が実行しても同じ結果が得られる.

スマートコントラクトは一般に Soldity [23] などの高級言語でソースコードとして記述される. このソースコードはコンパイルされブロックチェーン上にデプロイされたのち, ネットワーク上に存在するピアと呼ばれるノードによって管理され, 任意のピアで動作するグローバルかつ単一のインスタンスとして実行される. スマートコントラクトはコントラクトアドレスを識別として与えられ, このアドレスを通じて Ether の受け取りと関数の実行を行う. また, ピアにコントラクト関数を実行する利権を付与するため, Ethereum はガスと呼ばれる仮想通貨 Ether の支払燃料を設定している. ガスは計算の複雑さに依存し, 無限ループなどを防ぐ狙いがある.

スマートコントラクトの脆弱性は The DAO 事件に代表されるように, 高額となる金銭的被害を巻き起こす. このため, 脆弱性の対策が盛んに行われてきた [15]. とくにコンパイラを更新することで脆弱なコードのデプロイを防ぐようになっている. また, 2.2.2 節で述べるような脆弱性の解析ツールも盛んに行われている.

2.2 関連研究

関連研究として Ethereum スマートコントラクトに関する実態調査および Ethereum スマートコントラクトの脆弱性解析ツールを紹介する.

2.2.1 実態調査に関する研究

Ethereum スマートコントラクトの脆弱性に関する実態調査を行った研究を紹介する. 文献 [24] では脆弱なコントラクト及び実際に被害が発生した件数と被害額を調査している. 文献 [24] と本稿の結果を突合することで, コンパイラのバージョンごとの被害額の算出が期待できる.

一方, 文献 [17] では, 既存の脆弱性解析ツールに対する性能調査として, 2015 年から 2019 年までに作成されたスマートコントラクトの解析を行い, 脆弱性件数の経年変化

を調査した. また, 各脆弱性解析ツールの開発・公開状況などの調査もある [16], [19]. スマートコントラクトの開発者が実際に最新のコンパイラおよび脆弱性解析ツールを利用しているかの調査結果 [22] もあり, 半数程度の開発者しか最新のコンパイラおよびツールを利用していないことが示されている. 上述した実態調査は解析ツールの性能あるいは開発者観点での調査となっており, 実際に作成されたコントラクトがどの程度脆弱性を含むかは示されてこなかった. 一方で, 本稿の結果と合わせることで, 今後対策がされるべき脆弱性の指針などを示すことが可能である.

2.2.2 解析ツールの設計

Ethereum スマートコントラクトの安全性解析は, 未知の変数をシンボル変数として扱うことで解析する シンボリック実行 [4], [6], [14], [25], [26] が主流である. スマートコントラクトはプログラム外にあるブロックチェーンの情報を参照するため, スマートコントラクトの解析とシンボリック実行は相性が良い [7]. 近年は, コントラクト間での脆弱性解析やコントラクト生成への対応など, 解析範囲の拡大に注力している [7], [8], [10], [27]. スマートコントラクトの安全性解析におけるもうひとつの主要な手法は, プログラムが仕様を満たしているか述語論理により確認する形式検証 [12], [13], [28], [29], [30], [31] である. 近年では形式検証により脆弱性の汎用的な解析まで実現できている [9]. 最後に動的解析ツール [32], [33], [34], [35] は, コードを実際に実行・監視することで安全性解析を行う. 様々な攻撃を想定した汎用的な動的解析 [33] も知られている.

2.3 本研究の狙い

本稿では「Q1: スマートコントラクトの脆弱性はどのように変化しているか」, 「Q2: コンパイラのバージョンが各脆弱性にどう影響しているか」を明らかにする. これは, 脆弱性の実態調査として関連研究に不足していた観点である. 以下に, その狙いを詳細に述べる.

まず Q1 について, 実際にデプロイされているコントラクトを通じて, 各脆弱性の発生件数がどのように変化しているか調査する. 各脆弱性の合計数そのものは既存の実態調査 [17], [24] で示されている一方, 既存の実態調査では時系列に応じてどのように変化しているか示されていない. また, 脆弱性の発生時期は一般には予測できないことから, 既存成果の合計数から各時期ごとの変化を類推することも容易ではない. このため, 本稿ではまず各脆弱性の発生件数を, 時系列に従って確認する. この問いを明らかにすることで, 後続の研究開発がどの脆弱性に焦点を当てるべきか指針を示すことが可能となる. さらに, この問いを通じて, 次のコンパイラに関する問い (Q2) が創発される.

次に Q2 について, 本稿独自の観点としてコンパイラのバージョンに基づいた調査を検討する. 2.2 節で述べた通り, 最新のコンパイラの動作は不安定 [36], [37] などの理由で, 半数程度の開発者しか最新のコンパイラあるいは脆弱性解析ツールを利用していない [22]. これに対し, 本稿ではコンパイラのバージョンごとに各脆弱性の発生件数を確認する. これにより, 各コンパイラの更新による脆弱性への影響を明らかにすることで, プラットフォームレベルで対策されている脆弱性を示すことが期待できる. なお, 既存の実態調査はコンパイラについて考慮しておらず, その結果からコンパイラによる影響を推測することは難しい.

3. Ethereum スマートコントラクトの脆弱性調査

本節では前節で述べた問いに対する本稿の調査方法を述べる. まず脆弱性の解析ツールと対象とする脆弱性を述べたのち, 解析対象となるコントラクトの収集方法を述べる.

3.1 調査に用いるツールと脆弱性の定義

本稿では脆弱性の調査に SmartCheck [5] を用いる. SmartCheck [5] は主に Solidity で記述されたコントラクトを対象とした静的解析ツールである. 機能としてはコードの中間表現として XML 構文木を生成したのち, 脆弱性のパターンに相当する XQuery パス表現を構文木から探索する. コードが GitHub 上に公開されており, また, 解析精度が高いことが示されている [17], [18].

本稿では既存の脆弱性の実態調査 [24] に従い, ツールが脆弱と判定するコントラクトを脆弱性を持つコントラクトと定義する. これは既存の調査でも述べられている通り, 用いるツールの性能に従って偽陽性あるいは偽陰性が乗じる可能性もある. この観点については, 5.3 節で議論する.

3.2 対象とする脆弱性

本稿では, 重要度 [5] の高い脆弱性として知られる Using tx.origin, Unchecked Call, Locked Money 脆弱性を調査対象とする. これらの脆弱性には予備検討を踏まえて, 着目した. 以降では予備検討, および, 各脆弱性の詳細を述べる.

3.2.1 予備検討

調査に向けた予備検討として, ブロックチェーンに保存された使用履歴を検索し, 使用頻度の最も高い上位 124,639 件のコントラクトを収集し, そのうち, ソースコードが公開されており, かつ, Solidity によって記述されているコントラクト 77,372 件を解析した. 一般には脆弱性の重要度と発生した際の被害額は連動することから, それぞれのコントラクトが持つ脆弱性の中でも重要度が最も高い数値”3”である脆弱性のみに着目した. 該当するコントラクトを抽出し, 時系列に表示した結果が図 1 である. この図において各脆弱性の名称は, SmartCheck のリポジトリ記載の情報に従っている[*3].

[*3] https://github.com/smartdec/smartcheck/tree/master/rule_descriptions

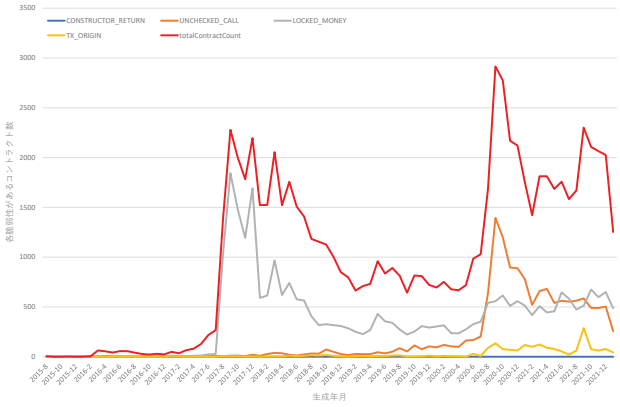

図 1　重要度 3 の脆弱性があるコントラクトの数

このうち Constructor Return はとくに生成されていないことが分かった．また，Re-entrancy は検知できるツール [4], [5], [6], [7], [9], [13], [25], [32], [38], [39] が多いことに加え，コードレベルでの対策も知られている[*4]．このため，それ以外の重要度の高い 3 種類に着目した検討を行う．

3.2.2　Unchecked Call 脆弱性

この脆弱性は mishandled exceptions という名前でも知られており，コントラクトが別のコントラクトを呼び出す場合に，呼び出し先コントラクトの例外を考慮しないことで生じる脆弱性である [4]．具体的に，コントラクト呼び出しに伴い送金を行う場合を考える．この場合，呼び出し先のコントラクトで例外が発生すると状態の巻き戻しが行われるため，通貨の受け取りは完了しない．しかし，呼び出し元のコントラクトでは例外処理を行わないため，通貨の受け取りは完了したことになる．これにより，送金されるべき通貨が失われてしまう状況が起こりえる．v0.4.10 では送金先のコントラクトで例外が発生した場合，呼び出し元でも例外を発生させる `transfer` が導入されており，送金時にはこの関数の使用が推奨される．

3.2.3　Locked Money 脆弱性

この脆弱性は，複数人による署名が必要となるトランザクションにおいてそのうちの一人が正しく署名を行わないことによって検証が通らず，それによって通貨が凍結してしまうパリティウォレット攻撃 [*5] に付随して発見された脆弱性である．本脆弱性は通貨を受け取るための `payable` 関数はあるが，送金するための関数，例えば `send` や `transfer` が存在しない場合の脆弱性であり，受け取った通貨がコントラクト内で凍結されてしまう．これを防ぐためには，通貨の受け取りを行う関数を実装しない，または，送金するための関数を実装する必要がある．

[*4] External Calls and Reentrancy: `https://docs.soliditylang.org/en/latest/smtchecker.html?highlight=reentrancy#external-calls-and-reentrancy`

[*5] `https://www.comae.com/posts/the-280m-ethereums-parity-bug./`

3.2.4　Using tx.origin 脆弱性

この脆弱性は，トランザクションの起点となったユーザのアドレスを表す Solidity 上のグローバル変数 `tx.origin` を使用することで引き起こされる [*6]．ただし，これはコンパイラ v0.5.0 より前と，それ以降で脆弱性の原因が少し異なる．まず，v0.5.0 より前のコンパイラを使用している場合，コンストラクタの呼び出し時に引数として攻撃者が自身のアドレスを指定することで `tx.origin` を書き換えることができてしまう．そのため，`tx.origin` を送金の認証に使用していた場合，意図せず攻撃者へ送金されてしまうような攻撃が可能となる．一方，v0.5.0 以降では仕様が変更され，宣言時の引数の数と呼び出し時の引数の数が異なる場合はエラーが発生する仕様となった [40]．これによって `tx.origin` の書き換えはできなくなっているが，`tx.origin` の使用自体が制限されたわけではない．このため，v0.5.0 以降においても注意すべき脆弱性である．

3.3　調査設定

Ethereum では，そのブロック探索用プラットフォームである Etherscan[*7]によって，ブロックチェーン上の情報を取得するための API が提供されている．本稿ではこの API を使用し，日本時間において 2022 年 4 月 30 日までにデプロイされた全コントラクトを脆弱性調査の対象として収集した．とくに，Solidity で記述されたソースコードが公開されている 626,171 件のコントラクトを収集し，解析を行なっている．なお，Solidity のソースコードがないコントラクトについては，今後の課題である．

コントラクトのソースコード収集の具体的な流れを以下に示す．(1) 2022 年 4 月 30 日までに生成されたブロックを順に参照する．(2) ブロック内に含まれるトランザクションのうち，その宛先が空になっているもの，つまりコントラクト作成トランザクションを検索し，そのトランザクションハッシュを取得する．(3) 取得したハッシュからトランザクションの情報を参照し，生成されたコントラクトのアドレスを取得する．(4) 取得したアドレスから，そのソースコードとコンパイラの種類とバージョン情報が公開されている場合は，それらの情報も取得する．

取得したソースコードに対し SmartCheck を実行し，脆弱性を持つか判定する．このとき，脆弱性を持つと判定されたコントラクトの数を発生数と呼び，また，以下の式で計算される数値を発生率と呼ぶ．

$$\text{発生率} = \frac{\text{発生数}}{\text{その月にデプロイされたコントラクト数}}$$

ここで，分母は前述した通り Solidity のコードが公開されているものを表す．本稿では脆弱性の評価指標に上述した発生数と発生率を用いる．

[*6] `https://vessenes.com/tx-origin-and-ethereum-oh-my/`

[*7] `https://etherscan.io/`

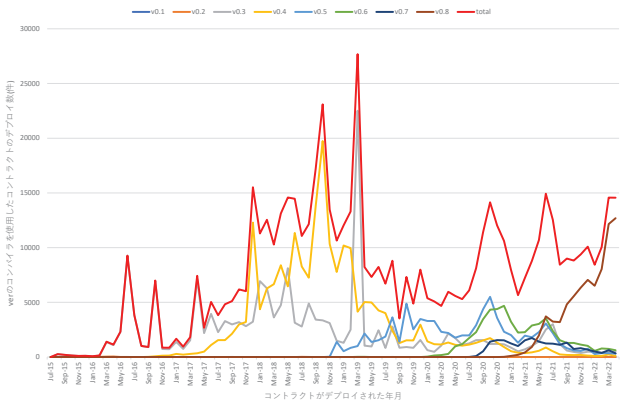

図 2　解析対象としたコントラクトの数

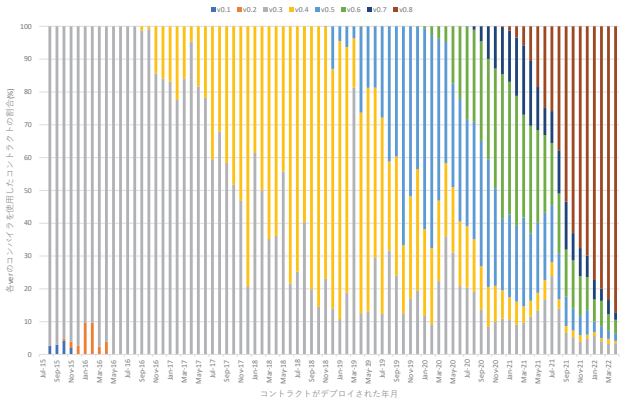

図 3　各コンパイラバージョンの使用率

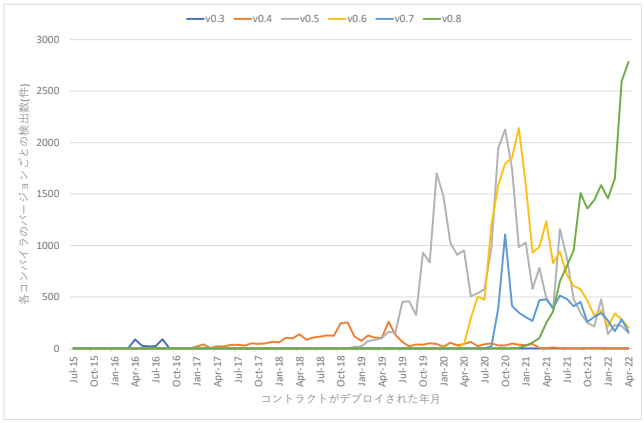

図 4　各コンパイラのバージョンごとの Unchecked Call の発生数

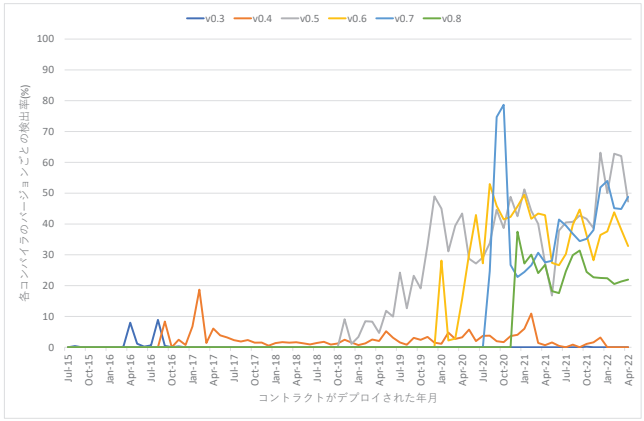

図 5　各コンパイラのバージョンごとの Unchecked Call の発生率

4.　調査結果

本節では, 前節で述べた設定に基づいて実施した脆弱性の調査結果を述べる. とくに各脆弱性について, コンパイラのバージョンごとの結果も踏まえて議論する. 具体的には, 各コンパイラのバージョンを使用したコントラクトに対して, そのうち脆弱性があるコントラクトがどれだけ存在するのかについての割合を月ごとに調査している. ただし, v0.1, v0.2 については, 解析対象となったコントラクトの数がそれぞれ 32 件, 127 件と少数であったため, これ以降は 0.3 以降のバージョンに対して議論する.

4.1　Unchecked Call

まず解析対象としたコントラクトについて, コンパイラの各バージョンを使用したコントラクトの数とそのコンパイラの使用率をそれぞれ図 2 と 図 3 に示す. このとき, コンパイラのバージョンごとの Unchecked Call の発生数を図 4, 発生率を図 5 に示す. 図 4 から最新バージョンである v0.8 で発生数を確認すると, その数は急激に増加している. しかし, これは図 2 に示されているようにデプロイされているコントラクトが増加していることが原因であり, その発生率は図 5 に示す通りわずかに減少傾向にあることが確認できる. 具体的には, v0.8 は 2022 年 4 月において, その時点で利用が確認されているコンパイラのバージョンの中では最も発生率が低くなっている.

4.2　Locked Money

各コンパイラのバージョンごとの Locked Money の発生数を図 6, 発生率を図 7 にそれぞれ示す. 図 7 によると, 各バージョン更新が行われてから半年ほど発生率が増加している傾向が見受けられる. ただし, 発生率はその後減少していることが図において v0.5, v0.6, および v0.8 で確認できる.

4.3　Using tx.origin

各コンパイラのバージョンごとの Using tx.origin の発生数を図 8, 発生率を図 9 に示す. Using tx.origin は, 図 9 からわかるように, この脆弱性があるコントラクトは少数である. しかし, いずれのバージョンにおいても 0 にまでは収束しておらず, また, v0.7 を除き 2021 年 10 月以降はいずれのバージョンも同じ程度の発生率となっている.

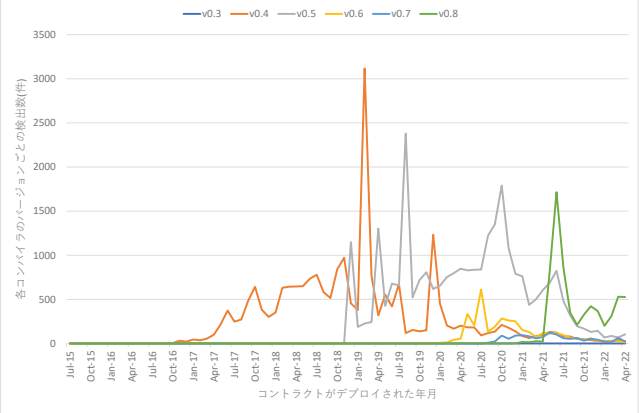

図 6 各コンパイラのバージョンごとの Locked Money の発生数

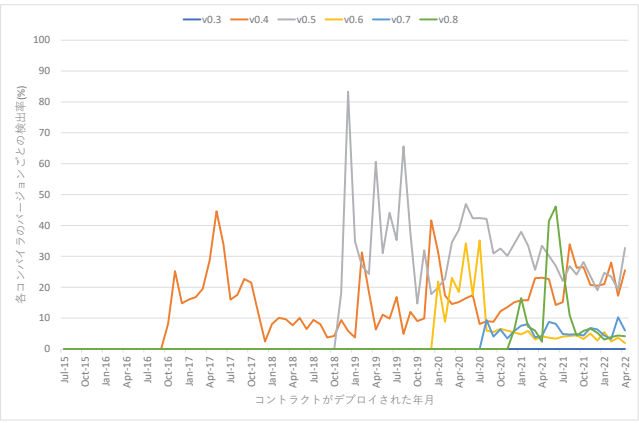

図 7 各コンパイラのバージョンごとの Locked Money の発生率

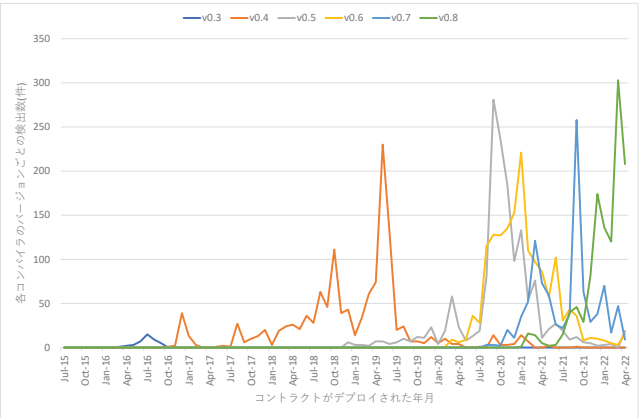

図 8 各コンパイラのバージョンごとの Using tx.origin の発生数

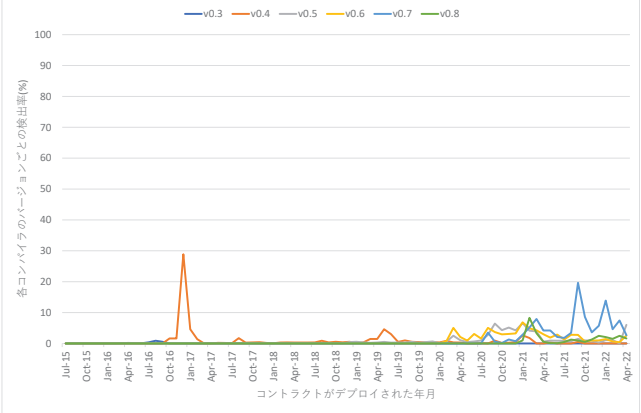

図 9 各コンパイラのバージョンごとの Using tx.origin の発生率

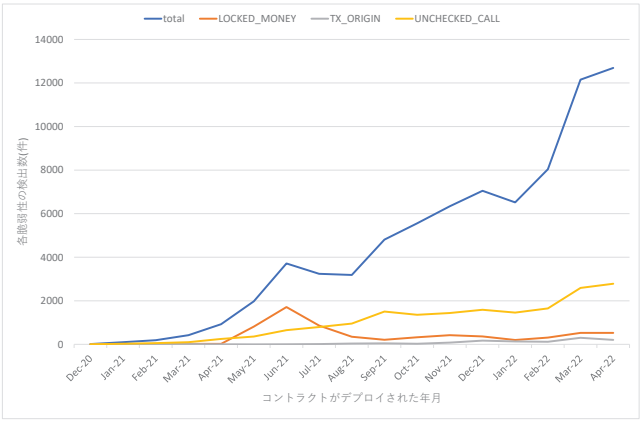

図 10 コンパイラ v0.8 でコンパイルされたコントラクトが持つ各脆弱性の発生数

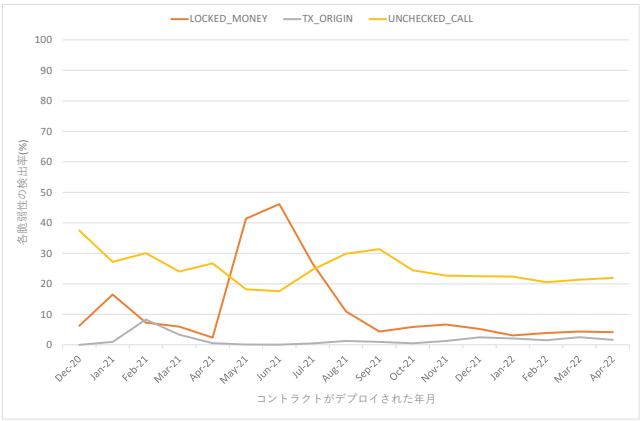

図 11 コンパイラ v0.8 でコンパイルされたコントラクトが持つ各脆弱性の発生率

5. 考察

5.1 コンパイラ更新の影響

前節で示した調査結果に基づいて, コンパイラの影響について考える. 具体的には, コンパイラの更新により脆弱性がどれだけ減少しているか, また, 更新による弊害について, それぞれ以下に考察する.

5.1.1 脆弱性の減少

調査結果において脆弱性を持つとされたコントラクトに関して, v0.8 から v0.3 までコンパイラのバージョンごとに各脆弱性における発生数および発生率を算出した. その結果を図 10 から図 21 までに示す.

発生数を表す図における total は, そのコンパイラバージョンを使用しているコントラクトの数を表している

また, 各バージョンにおいてコントラクトが最も多くデプロイされた月における各脆弱性の発生率を表 1 に示す.

表 1 に示す結果によると, v0.6 から v0.8 への更新にか

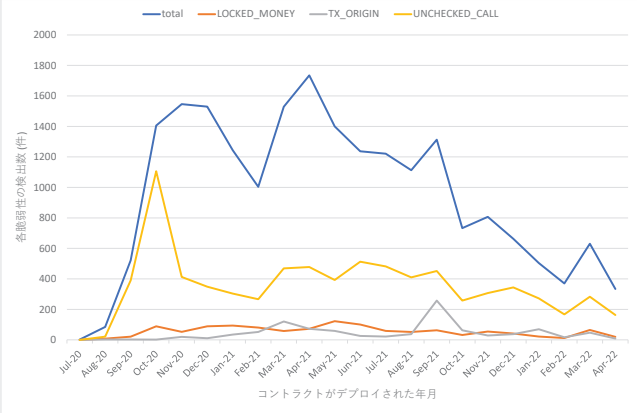

図 12 コンパイラ v0.7 でコンパイルされたコントラクトが持つ各脆弱性の発生数

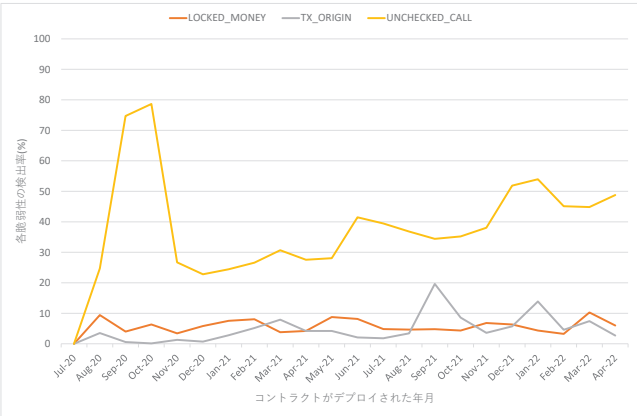

図 13 コンパイラ v0.7 でコンパイルされたコントラクトが持つ各脆弱性の発生率

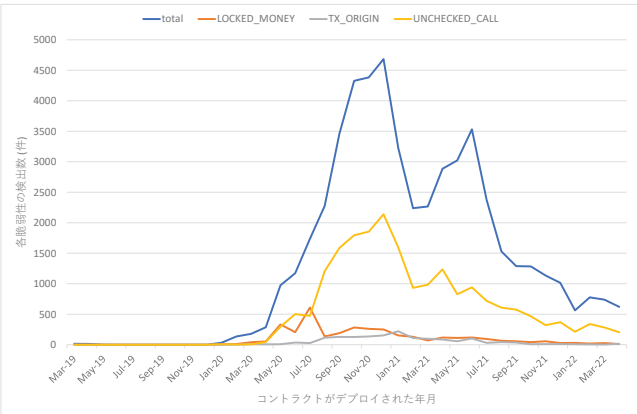

図 14 コンパイラ v0.6 でコンパイルされたコントラクトが持つ各脆弱性の発生数

けて脆弱性の発生率が（v0.7 における Using tx.origin を除き）一貫して低下している．とくに v0.7 から v0.8 への更新で，とくに v0.8 では解析したコントラクト数が多いにもかかわらず，Using tx.origin が 4.21%から 1.64%まで減少している．このため，コンパイラのバージョン更新による対策が有効であると考えられる．

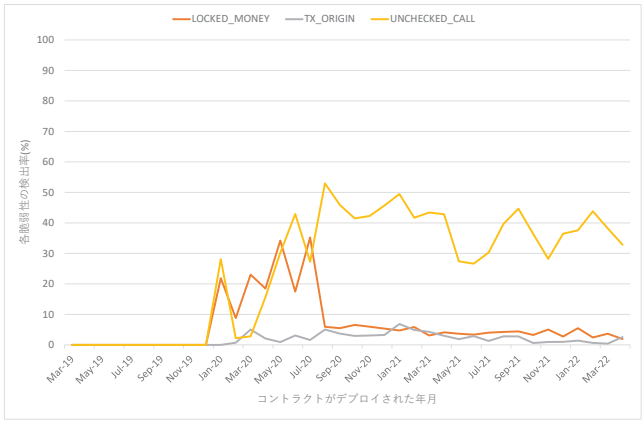

図 15 コンパイラ v0.6 でコンパイルされたコントラクトが持つ各脆弱性の発生率

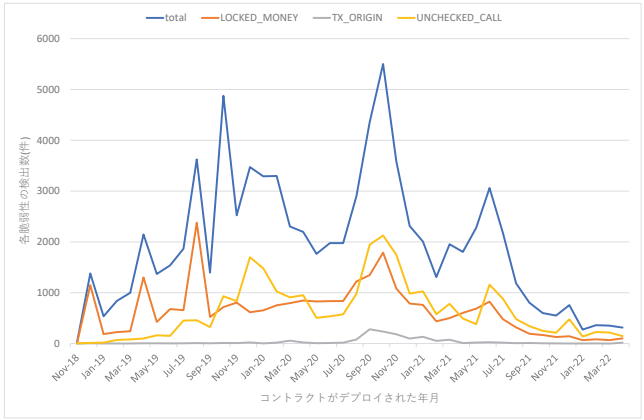

図 16 コンパイラ v0.5 でコンパイルされたコントラクトが持つ各脆弱性の発生数

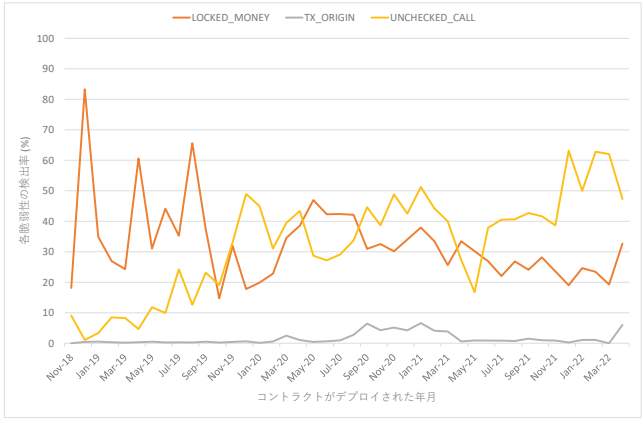

図 17 コンパイラ v0.5 でコンパイルされたコントラクトが持つ各脆弱性の発生率

5.1.2 更新の弊害

対策が有効であると考えられる一方で，コンパイラのバージョン更新による弊害も確認できる．具体的には，更新後数か月間は高い脆弱性の発生率にあり，その後収束していくバージョン更新が多く存在する．例として，v0.8 は 2020 年 12 月に開発され，図 11 からわかるように，2021 年 5 月から 7 月にかけて Locked Money の発生率が上昇してい

表 1　コントラクト生成数最大月における脆弱性の発生率

バージョン	解析したコントラクト (年月)	Locked Money	Using tx.origin	Unchecked Call
v0.8	12691 件 (2022/04)	4.14%(526 件)	1.64%(20 件)	21.34%(2785 件)
v0.7	1735 件 (2021/04)	4.21%(73 件)	4.21%(73 件)	27.55%(478 件)
v0.6	4683 件 (2020/12)	5.36%(251 件)	3.27%(153 件)	45.70%(2140 件)
v0.5	5500 件 (2020/10)	32.53%(1789 件)	4.29%(236 件)	38.67%(2127 件)
v0.4	19747 件 (2018/10)	4.25%(840 件)	0.56%(111 件)	1.24%(245 件)
v0.3	22516 件 (2019/03)	0%(0 件)	0%(0 件)	0%(0 件)

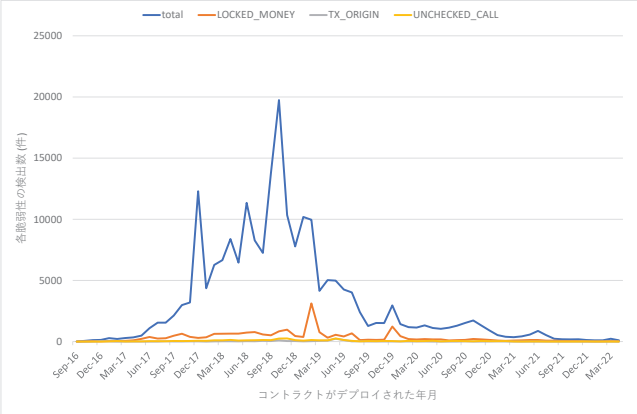

図 18　コンパイラ v0.4 でコンパイルされたコントラクトが持つ各脆弱性の発生数

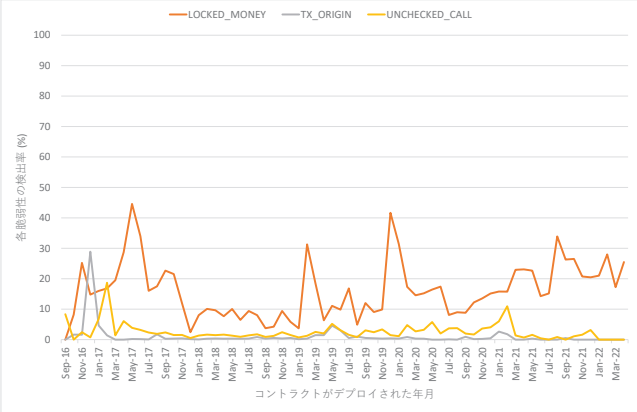

図 19　コンパイラ v0.4 でコンパイルされたコントラクトが持つ各脆弱性の発生率

る．また，v0.7 の図 13 においては，2020 年 7 月の開発後から 2020 年 11 月ごろまで，Unchecked Call の発生率が大きく上昇している．その理由としては，バージョン更新に潜在するバグによって動作が不安定になっていると考えられる．実際に過去の更新としては v0.5 にすることでその前の版より不安定になったこと [37]，また，開発者が最新のコンパイラでは動作が不安定になることを懸念している結果 [22] もある．その一方で，v0.8 の Locked Money と v0.7 の Unchecked Call の発生率は，開発後から半年程度して動作が安定している様子がうかがえる．そのため，バージョン更新直後は注意が必要であるが，半年後を目安に新しいバージョンのコンパイラを使用することが推奨される．

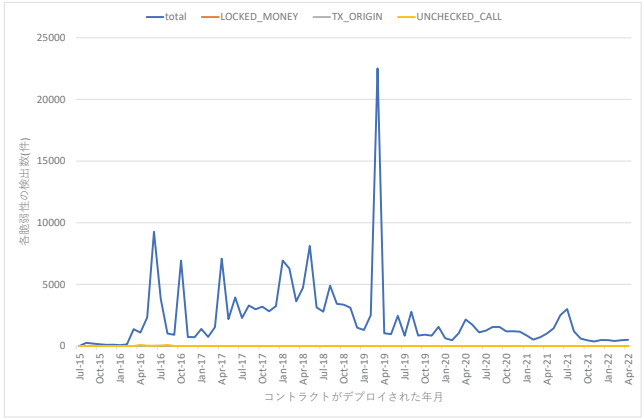

図 20　コンパイラ v0.3 でコンパイルされたコントラクトが持つ各脆弱性の発生数

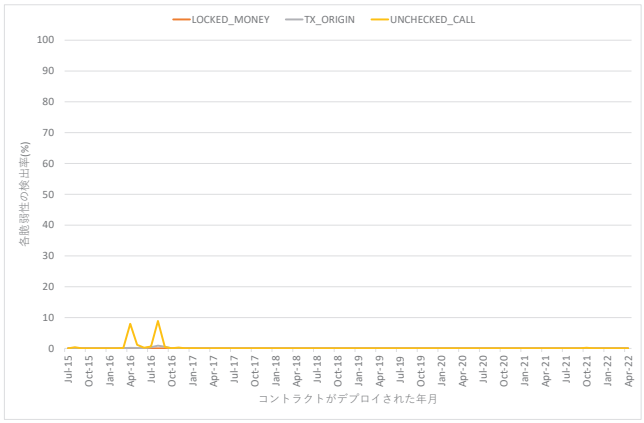

図 21　コンパイラ v0.3 でコンパイルされたコントラクトが持つ各脆弱性の発生率

5.2　Unchecked Call への対応

表 1 に示す結果から，Unchecked Call は v0.5 での 38.57% と比較して，v0.8 では 21.34% まで減少していることが確認できる．しかしその一方で，ほかの脆弱性と比較してその発生数は多く，発生率の減少傾向も緩やかである．このため，Unchecked Call に対しては今後のさらなる対策が必要である．以下では，Unchecked Call の発生数が未だ多い原因について考察する．

Unchecked Call の発生数が多い理由について，実際に SmartCheck により脆弱性を持つと検出されたコードを確認したところ，複数のファイルで使用されているライブラリ中に，戻り値を確認していない `delegatecall` 関数 [*8] が

[*8] `https://github.com/OpenZeppelin/`

```
1 function _functionDelegateCall(address target,
      bytes memory data) private returns (bytes
      memory) {
2     require(AddressUpgradeable.isContract(
          target), "Address: delegate call to
          non-contract");
3
4     // solhint-disable-next-line avoid-low-
          level-calls
5     (bool success, bytes memory returndata) =
          target.delegatecall(data);
6     return AddressUpgradeable.verifyCallResult
          (success, returndata, "Address: low-
          level delegate call failed");
7 }
```

Listing 1 再利用されているコードの例

存在した．この delegatecall 関数が脆弱性として検知されていた．そのコードを Listing 1 に示す．

この点を踏まえ，Listing 1 をはじめとする脆弱性を持つコードの再利用が，Unchecked Call の発生率が高い原因と仮説を立てた．実はスマートコントラクトではコードの再利用や使いまわしが多いことが指摘されている [41]．とくにコンパイラのバージョンが上がるにつれ再利用が増加しており，v0.4 では 49%の再利用があることに対し，v0.7 では 82%の再利用がある [42]．つまり，脆弱性が検知されたコントラクトの中には，脆弱性を持つ既存のコントラクトを再利用したものが多くあることが予想される．このため，既存の Unchecked Call 脆弱性を持つコントラクトが使いまわされていると考えられる．

本稿では各コントラクトが新たに生成されたものか，いずれかのコードを再利用したものかは考慮していない．コードの再利用の可能性を考慮し，新たに作成されたコントラクトのみを対象とすることで，より正確にコンパイラのバージョンによる脆弱性への影響が考慮できると考えられる．この再利用を考慮した調査は，今後の課題である．

5.3 制約条件

本稿での調査は対象とするコード，および，偽陽性と偽陰性の 2 点が制約条件である．これらを検討することが今後の課題である．以下にその詳細を述べる．

対象とするコード: 本研究では，使用した解析ツールの解析対象である Solidity のソースコードが公開されているコントラクトのみを取得し，解析を行なった．近年では Solidity 以外にも Vyper など高級言語が様々開発されており，また，インラインアセンブリを導入した開発などもされている [43]．今後は EVM バイトコードを対象とするなど，記述言語に依らないような解析も必要である．

偽陽性と偽陰性: 脆弱性の調査に SmartCheck のみを用いた．これは SmartCheck の精度に解析結果が依存することを意味している．一方，文献 [24] によると，少なくとも偽陰性は用いる解析ツールを増やすこと，また，いずれかのツールが脆弱と判定したら脆弱性を持つコントラクトと区分することで，可能な限り低減することが可能である．現実的な脅威を鑑みると，偽陰性は偽陽性よりも重要であることから，今後は SmartCheck 以外の解析ツール [4], [6], [9], [13], [14] も導入することで，偽陰性を可能な限り低減した検討も行う．

6. 結論

本稿では，Ethereum スマートコントラクトにおいて脆弱性があるプログラムがどれだけ作成されているか，コンパイラのバージョンごとに調査した．調査としては 2022 年 4 月までに Ethereum ブロックチェーンに保存されたコントラクトのうち，Solidity のソースコードが公開されているコントラクト 626,171 件を収集し，脆弱性解析ツール SmartCheck で解析している．結果として，重要度の高い脆弱性である Unchecked Call, Locked Money, Using tx.origin が，コンパイラのバージョン更新により，脆弱性の発生率が減少していることを確認した．また，コンパイラ更新により，更新直後に一時的に脆弱性の発生数が増加すること，また，その弊害が半年程で安定することを確認している．このため，少なくともコンパイラを更新してから半年後には，更新したコンパイラを使うほうが望ましいといえる．なお，Unchecked Call 脆弱性は減少しているとはいえ，まだ発生率が 21.34% と高く，今後の対策が望まれる．

今後の課題は，EVM バイトコードを対象とすることでより広範囲の調査，および，既存の評価 [17], [18] で精度が高いツールとして知られる Mythril [14] などを用いたより高精度の調査をそれぞれ実施することである．また，コードの再利用を考慮することで，真の意味で新たに生成されたコントラクトのみを対象とした調査も実施予定である．

謝辞

本研究の一部は株式会社野村アセットマネジメントとの共同研究，および，文部科学省による Society 5.0 実現化研究拠点支援事業 (グラント番号: JPMXP0518071489) によって実施されている．また，データ分析に助言いただいた大阪大学セキュリティ工学講座・後藤勇芽輝氏に感謝する．

参考文献

[1] Wood, G.: Ethereum: A secure decentralised generalised transaction ledger Byzantium Version, https://ethereum.github.io/yellowpaper/paper.pdf.
[2] Zou, W. et al.: Smart Contract Development: Challenges and Opportunities, *IEEE Transactions on Soft-*

openzeppelin-contracts-upgradeable/blob/master/contracts/proxy/ERC1967/ERC1967UpgradeUpgradeable.sol

ware Engineering, pp. 1–1 (2019).

[3] 祐一郎, 知., 奈実, 芦., 直人, 矢. and ポール, ク. ジ.: スマートコントラクト――ブロックチェーンからなるプログラミングプラットホーム――, **電子情報通信学会　通信ソサイエティマガジン**, Vol. 14, No. 1, pp. 26–33 (online), DOI: 10.1587/bplus.14.26 (2020).

[4] 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).

[5] Tikhomirov, S., Voskresenskaya, E., Ivanitskiy, I., Takhaviev, R., Marchenko, E. and Alexandrov, Y.: Smartcheck: Static analysis of ethereum smart contracts, *Proc. of WETSEB 2018*, ACM, pp. 9–16 (2018).

[6] Feist, J., Grieco, G. and Groce, A.: Slither: A Static Analysis Framework for Smart Contracts, *Proc. of WETSEB 2019*, IEEE, pp. 8–15 (2019).

[7] Chinen, Y., Yanai, N., Cruz, J. P. and Okamura, S.: Hunting for Re-Entrancy Attacks in Ethereum Smart Contracts via Static Analysis, *arXiv preprint arXiv:2007.01029* (2020).

[8] Frank, J., Aschermann, C. and Holz, T.: ETHBMC: A Bounded Model Checker for Smart Contracts, *Proc. of Usenix Security 2020*, USENIX Association, pp. 2757–2774 (2020).

[9] Schneidewind, C., Grishchenko, I., Scherer, M. and Maffei, M.: EThor: Practical and Provably Sound Static Analysis of Ethereum Smart Contracts, *Proc. of CCS 2020*, ACM, p. 621–640 (2020).

[10] Weiss, K. and Schütte, J.: Annotary: A Concolic Execution System for Developing Secure Smart Contracts, *Proc. of ESORICS 2019*, LNCS, Vol. 11735, Springer, pp. 747–766 (2019).

[11] Ashizawa, N., Yanai, N., Cruz, J. P. and Okamura, S.: Eth2Vec: Learning Contract-Wide Code Representations for Vulnerability Detection on Ethereum Smart Contracts, *Proc. of BSCI 2021*, ACM, pp. 47–59 (online), DOI: 10.1145/3457337.3457841 (2021).

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

[13] 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).

[14] Mueller, B.: Smashing Ethereum Smart Contracts for Fun and Real Profit, *9th HITB Security Conference* (2018).

[15] Chen, H., Pendleton, M., Njilla, L. and Xu, S.: A Survey on Ethereum Systems Security: Vulnerabilities, Attacks, and Defenses, *ACM Computing Surveys*, Vol. 53, No. 3 (2020).

[16] Kushwaha, S. S., Joshi, S., Singh, D., Kaur, M. and Lee, H.-N.: Systematic Review of Security Vulnerabilities in Ethereum Blockchain Smart Contract, *IEEE Access*, Vol. 10, pp. 6605–6621 (online), DOI: 10.1109/ACCESS.2021.3140091 (2022).

[17] Durieux, T., Ferreira, J. F., Abreu, R. and Cruz, P.: Empirical review of automated analysis tools on 47,587 Ethereum smart contracts, *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (2020).

[18] Parizi, R. M., Dehghantanha, A., Choo, K.-K. R. and Singh, A.: Empirical Vulnerability Analysis of Automated Smart Contracts Security Testing on Blockchains, *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering*, IBM Corp., p. 103–113 (2018).

[19] Angelo, M. D. and Salzer, G.: A Survey of Tools for Analyzing Ethereum Smart Contracts, *2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)*, pp. 69–78 (2019).

[20] Kushwaha, S. S., Joshi, S., Singh, D., Kaur, M. and Lee, H.-N.: Ethereum Smart Contract Analysis Tools: A Systematic Review, *IEEE Access*, pp. 1–1 (Early Access) (2022).

[21] Di Angelo, M. and Salzer, G.: A survey of tools for analyzing ethereum smart contracts, *Proc. of DAPPCON 2019*, IEEE, pp. 69–78 (2019).

[22] Wan, Z., Xia, X., Lo, D., Chen, J., Luo, X. and Yang, X.: Smart Contract Security: A Practitioners' Perspective, *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, IEEE/ACM, pp. 1410–1422 (2021).

[23] v0.8.0 Breaking Changes, S.: https://docs.soliditylang.org/en/latest/080-breaking-changes.html.

[24] Perez, D. and Livshits, B.: Smart Contract Vulnerabilities: Vulnerable Does Not Imply Exploited, *Proc. of USENIX Security 21)*, USENIX Association, pp. 1325–1341 (2021).

[25] 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).

[26] 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).

[27] Permenev, A., Dimitrov, D., Tsankov, P., Drachsler-Cohen, D. and Vechev, M.: Verx: Safety verification of smart contracts, *Proc. of IEEE S&P 2020*, IEEE, pp. 414–430 (2020).

[28] 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).

[29] 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).

[30] 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).

[31] Grishchenko, I., Maffei, M. and Schneidewind, C.: Foundations and Tools for the Static Analysis of Ethereum Smart Contracts, *Proc. of CAV 2018*, LNCS, Vol. 10981, Springer, pp. 51–78 (2018).

[32] 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).

[33] Ferreira Torres, C., Steichen, M., Norvill, R., Fiz Pontiveros, B. and Jonker, H.: ÆGIS: Shielding Vulnerable Smart Contracts Against Attacks, *Proc. of AsiaCCS 2020*, ACM (2020).

[34] Ferreira Torres, C., Iannillo, A. K., Gervais, A. and State, R.: The Eye of Horus: Spotting and Analyzing Attacks on Ethereum Smart Contracts, *Financial Cryptog-*

raphy and Data Security, LNCS, Vol. 12674, Springer, pp. 33–52 (2021).

[35] Chen, T., Cao, R., Li, T., Luo, X., Gu, G., Zhang, Y., Liao, Z., Zhu, H., Chen, G., He, Z., Tang, Y., Lin, X. and Zhang, X.: SODA: A Generic Online Detection Framework for Smart Contracts, *Proc. of NDSS 2020*, Internet Society (2020).

[36] Tantikul., P. and Ngamsuriyaroj., S.: Exploring Vulnerabilities in Solidity Smart Contract, *Proc. of ICISSP 2020*, INSTICC, SciTePress, pp. 317–324 (online), DOI: 10.5220/0008909803170324 (2020).

[37] Crafa, S. and Pirro, M. D.: Solidity 0.5: when typed does not mean type safe, *arXiv preprint arXiv:1907.02952* (2019).

[38] Qian, P., Liu, Z., He, Q., Zimmermann, R. and Wang, X.: Towards Automated Reentrancy Detection for Smart Contracts Based on Sequential Models, *IEEE Access*, Vol. 8, pp. 19685–19695 (2020).

[39] 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).

[40] v0.5.0 Breaking Changes, S.: https://docs.soliditylang.org/en/latest/050-breaking-changes.html.

[41] Chen, X., Liao, P., Zhang, Y., Huang, Y. and Zheng, Z.: Understanding Code Reuse in Smart Contracts, *Proc. of SANER 2021*, IEEE, pp. 470–479 (2021).

[42] Pierro, G. A. and Tonelli, R.: Analysis of Source Code Duplication in Ethreum Smart Contracts, *Proc. of SANER 2021*, IEEE, pp. 701–707 (2021).

[43] Liao, Z., Song, S., Zhu, H., Luo, X., He, Z., Jiang, R., Chen, T., Chen, J., Zhang, T. and Zhang, X.-s.: Large-Scale Empirical Study of Inline Assembly on 7.6 Million Ethereum Smart Contracts, *IEEE Transactions on Software Engineering*, pp. 1–1 (Early Access (2022).