Smart Contract Development Best Practices

ELEN4010 - Software Development III 2018/05/07

Christopher Maree - 1101946

Abstract— **This paper outlines and compares the current research into Ethereum smart contract development best practice relating to contract security. Previously hacked contracts are identified and discussed, breaking down exactly how the hack occured. A selection of the latest symbolic execution-based and static analysis tools are then used to analyze these contracts to try and identify the exploited vulnerabilities. Additional analysis tools are applied to these contracts, such as linters. Three popular Ethereum code bases are then analysed, over a series of releases, with two different Solidity linters, to generate sets of code metrics. These metrics can show a quantifiable improvement in contract quality over the development life cycle of the projects. Lastly, this report aims to provide a general guideline for smart contract security testing best practices, outlining a streamlined approach aimed at making contract development safer.**

I. Introduction

Ethereum smart contracts collectively manage billions of dollars in digital assets. The security of these contracts is of extreme importance as the immutable nature of the blockchain means that once a contract is deployed, it can't be changed. Previous smart contract hacks, such as TheDAO in 2016, resulted in the theft of 3.6 million Ether, valued at 72 million USD at the time of the hack (or 4.6 billion USD at the current all time high of 1280 USD). Clearly, the future security of the platform is vital to its success and adoption as the security of blockchain technology is one of its key features.

II. Technological Context

A brief contextual understanding of the underpinning technology is important to understand the inner workings of the hacks preformed and associated implications. The discussion of the underlying technologies is focused primarily on the security of the platforms rather than an indepth explanation of the technologies implementation.

A. Blockchain

A Blockchain is a public, peer-to-peer, digital ledger, used to record transactions chronologically. It represents an immutable, persistent record that can never be altered or changed by anyone due to the distributed nature of the consensus algorithm. Miners, all around the world, offer up their computation power to verify transactions (in "blocks") to secure the network in exchange for a fee. This is known as Proof of Work consensus [1]. It is this distribution of computation power, used in the verification of blocks, that results in the inherent security of the network; if one wants to attack it, they would require to have more than half of the total hash power on the network, commonly known as a 51% attack [2].

Note that there are other consensus algorithms, such as Proof of Stake, but the underlying concepts remain the same.

Blockchain was first proposed in 2008 by Satoshi Nakamoto. He created a "Peer-to-Peer Electronic Cash System" known as Bitcoin, based on blockchain technology [3]. It's invention was "hailed as a radical development in money and currency, being the first example of a digital asset which simultaneously has no backing or intrinsic value and no centralized issuer or controller" [4]. Digital cash is, however, only one of hundreds of potential use cases of blockchain technology.

Other uses of blockchain include digital assets to represent custom currencies, such as ETF's or other financial instruments, proof of ownership of physical property and ownership of non-fungible assets like domain names [5]–[7]. More complex implementations involve the control of digital assets through code enforcing pre-programed rules, known as smart contracts [8]. These smart contracts can be used to create all the above example implementations, as well as hundreds more.

B. Ethereum

Ethereum aims to create the ultimate protocol for building decentralised applications. Ethereum implements a Turing-complete programing language built into its blockchain, enabling anyone to create and deploy smart contracts [4]. The underlying technology is similar in principle to that of Bitcoin in that they are both run on public, distributed blockchain networks. There are clearly major technical differences between the two, enabling Ethereum's smart contract platform. Bitcoin does provide a protocol for simple scripting without the need for any extensions. It is however lacking in functionality and usability, without Turing-completeness as well as other fundamental limitations preventing its use in many smart contract applications [4].

Ethereum is built to be as *simple* as possible, even at the expense of inefficiencies. The technology aims to have a low barrier to entry, enabling intermediate programmers to utilize the technology easily. Ethereum aims to be *universal*, providing a platform on which other applications can be built through its internal Turing-complete scripting language. This enables any programmer to create their own smart contract and implementation on the protocol. The protocol is made to be *modular*, enabling simple upgradability without dependencies. The development process aims to be agile, enabling changes in the design as new discoveries are made. Lastly, it is *non-discriminatory*, never actively restricting access to its usage [4].

Ethereum can be thought of as a decentralised, world computer, disrupting the current client-server model [9]. Smart contracts are compiled to a low-level, stack-based bytecode language that is then run within the Ethereum Virtual Machine(EVM) [4]. This virtual machine can be viewed as a global computer, where each node runs the same execution of smart contract code to ensure consensus.

C. *Smart Contracts*

Smart contracts for Ethereum can be written in a multitude of languages that are all compiled down to Ethereum bytecode, the stack-based language, executed by the EVM [10]. The most popular language for writing smart contracts is Solidity, a "high-level imperative statically typed language compiled to EVM" [11]. Solidity is considered "contract-oriented" and its implementation is similar to C++, Python and Javascript [11].

For the purposes of this paper, in an attempt to keep the conceptual elements of blockchain simple, smart contracts can be thought of as autonomous programs that execute automatically when specific predefined conditions are met [12].

D. *Justification for Automated Smart Contract Audits*

The correct execution of smart contract code is always guaranteed by the blockchain's consensus protocol [13]. Despite this, programming errors within the contracts themselves can result in security vulnerabilities. It is up to the smart contract developer to ensure that the execution of the contract's *correctness* (utilizing appropriate syntax and best practice) and that the contract is *fair* (contract must abide by the predefined business logic upon execution) [13]. "Turing-completeness consistently leads to vulnerabilities, as a contract is naturally only as good as its developer." [14]

Manual smart contract auditing is possible but it is labor intensive and prone to errors. The hacked contracts discussed later on were all extensively manually audited by the best smart contract engineers before deployment. Despite this, they were still hacked, resulting in the loss of millions of USD worth of crypto assets. It is for this reason that automated tests are of utmost importance to provide formal verification of smart contracts to ensure correctness and fairness [13].

III. Current Security Research Overview

Given the pressing need for security within the Ethereum ecosystem, due to the multiple billions of dollars worth of digital assets at stake, a number of academic papers have been written outlining processes for smart contract security audits. Most prior research addresses security and privacy in designing smart contracts but few analyze smart contracts for vulnerabilities [13]. Some papers provide open source security audit tools for running security verification [13], [15]–[17]. Additionally, there are a myriad of open source projects aiming to improve security of smart contracts through static analysis [18]–[21]. There are also a collection of linting tools available, all capable of improving code quality by enforcing styling and composition rules [22], [23].

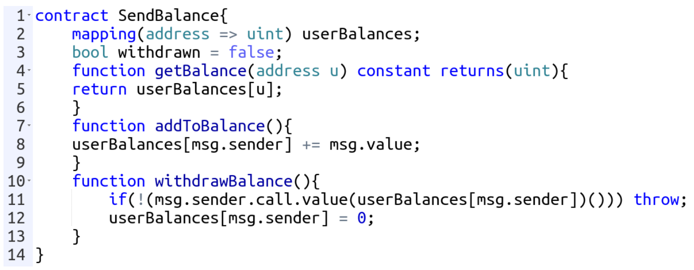
IV. Real Vulnerabilities Resulting in Lost Funds

The primary works in this field are broken down in turn, grouped into major sections based on the kind of exploit that the tool can identify and associated historically hacked contracts. Not all exploits identified in these works above will be discussed, but rather key examples where actual loss of funds occurred due to the hacks, namely theDAO, and both Parity hacks. These hacks were chosen as they had the largest effect on the community at large due to the wide spread effect of the exploits.

The contract examples presented in this report are simplified from the actual exploited contracts, aiming to show the underlying exploit mechanism. The full contracts that were exploited can be found on Github, here.

A. *Contract Reentrancy Vulnerability*

A reentrancy vulnerability was used to exploit the TheDAO, resulting in the theft of 3.6 million Ether, valued at 60 million USD at the time of the hack in 2016 [24]. Inter contract calls within Ethereum (calls from one contract to another) require the current execution to wait until the call is finished [25]. This opens the contracts up to a vulnerability whereby an attacker can exploit this intermediate state while the inter contract call finishes. A simplified version of this attack is shown in the code snippet below in 1.



```
contract SendBalance{
    mapping(address => uint) userBalances;
    bool withdrawn = false;
    function getBalance(address u) constant returns(uint){
    return userBalances[u];
    }
    function addToBalance(){
    userBalances[msg.sender] += msg.value;
    }
    function withdrawBalance(){
        if(!(msg.sender.call.value(userBalances[msg.sender])())) throw;
        userBalances[msg.sender] = 0;
    }
}
```

Fig. 1: Reentrancy Vulnerability Example

In this example, the contract that calls the SendBalance contract can call the withdrawBalance function again as the userBalances value has not yet been set to zero during the delay due to the inter contract call. This vulnerability occurs as the userBalances is only set to zero after the call is made and so the userBalances persistence storage remains non-zero during this delay, opening the contract up to being called recursively during this period. The attacker can call the withdrawBalance recursively, draining the contract of its Ether.

A simple fix to this exploit is too set the persistent storage userBalances to zero before executing the inter contract call, thereby removing the ability to call the contract recursively.

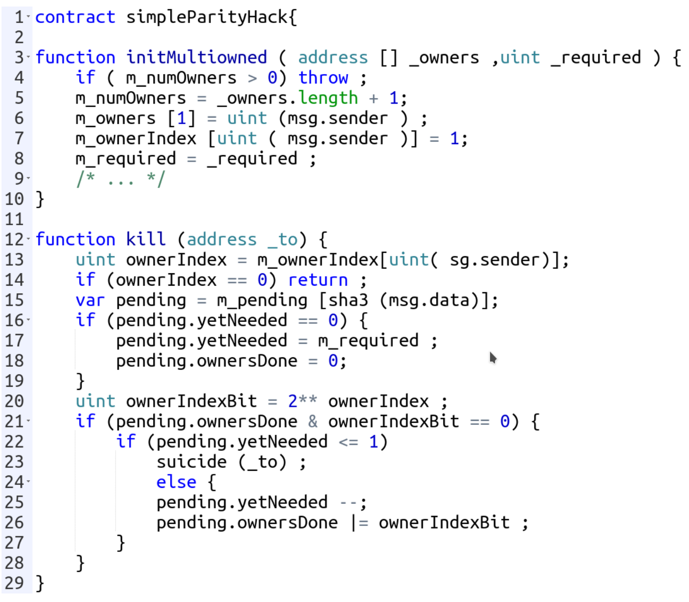
Oyente, a symbolic execution tool used to identify security vulnerabilities within contracts, is capable of identifying this reentrancy exploit within smart contracts [15]. Oyente is also capable of identifying bugs related to mishandled exceptions, timestamp dependencies and transaction-ordering dependence [15]. Oyente is implemented in Python and utilizes Z3 to define satisfiability [15]. Through simulation of the EVM, Oyente is able to identify exploits. It then utilizes a validator to remove false positive results by running the contracts on a private fork of the Ethereum blockchain, run locally within the Oyente test application.

TheDAO hack was so severe that the Ethereum blockchain forked to recover the lost funds from the hack, effectively rolling back the immutable blockchain to a point in time before the exploit was executed. This is the only time in the history of Ethereum that the blockchain was altered, and it was a major source of contention in the community as the blockchain is meant to remain immutable, unable for anyone to ever change it. There was however adequate consensus within the community to perform the hard fork.

B. *Incorrect Contract Modifier Vulnerability*

In Solidity, functions with no defined modifier are defaulted to public [11]. This is acceptable in most contexts, as long as the developer is judicious enough to ensure that functions that should not be public facing are internal or private.

The Parity enhanced-wallet, used in the creation of multi-signature wallets (wallets that require the signing of more than one private key to enable fund transfers, providing co-ownership over a wallet), was exploited by leaving key internal logic public. This exploit resulted in the theft of 150,000 Ether, valued at 30 million USD at the time of the hack [26]. Other multi-signature wallets were also found to be vulnerable but remaining funds were saved by a group of white hat hackers who rescued the vulnerable wallets before the attacker could drain them, saving 377,000 Ether, valued at 75.4 million USD at the time [27]. The code example below in 2 outlines the vulnerable part of the parity wallet that was exploited.



```
contract simpleParityHack{

function initMultiowned ( address [] _owners ,uint _required ) {
    if ( m_numOwners > 0) throw ;
    m_numOwners = _owners.length + 1;
    m_owners [1] = uint (msg.sender ) ;
    m_ownerIndex [uint ( msg.sender )] = 1;
    m_required = _required ;
    /* ... */
}

function kill (address _to) {
    uint ownerIndex = m_ownerIndex[uint( sg.sender)];
    if (ownerIndex == 0) return ;
    var pending = m_pending [sha3 (msg.data)];
    if (pending.yetNeeded == 0) {
        pending.yetNeeded = m_required ;
        pending.ownersDone = 0;
    }
    uint ownerIndexBit = 2** ownerIndex ;
    if (pending.ownersDone & ownerIndexBit == 0) {
        if (pending.yetNeeded <= 1)
            suicide (_to) ;
            else {
            pending.yetNeeded --;
            pending.ownersDone |= ownerIndexBit ;
        }
    }
}
```

Fig. 2: Parity Multisig Vulnerable Contract

Here, the initMultiowned function is intended to be used to initialize the wallet, defining who the owners are, the number required to sign a transaction and the total withdrawal limit per day allowed for the wallet. Due to the lack of the visibility modifier on the function, the function defaults to public, enabling anyone to call it, even after the wallet has been initialized. This enabled the attacker to call this function, setting themselves as the sole owner of the contract, defining the required number to one and setting the day limit high enough to drain the whole contract in one single withdrawal.

The fix to this exploit is trivially simple: setting the function to be marked as internal, preventing any outside party from calling this function and claiming ownership over the contract.

Due to the simplistic nature of this attack, most tools can identify the potential security vulnerability of this exploit by merely identifying that no access modifier was specified for the initMultiowned function. A linter, for example Solhint, can directly identify that this contract has a function without a defined access modifier, defaulting to public. Solhint informs the user of this potential error, ensuring that the developer checks if they want this function to indeed be public. Solhint also provides integration with a number of popular IDE's, such as Sublime Text 3, Atom and Vim to name a few [23]. This linting within the IDE could have prevented this exploit from happening even before the code was ever run for testing. The error should have been identified from within the IDE and addressed then and there.

Despite the simplicity of the hack and its easy mitigation, the bug remained dormant for 7 months and 4 days before being exploited [28]. This just goes to show how important it is to run continuous and thorough audits of all smart contracts.

C. Suicidal Contract Vulnerability

Ethereum contracts enable a security fallback feature with the `selfdestruct` function that removes the code from the blockchain [11]. The remaining Ether in the contract is sent to the designated target and the contracts storage and code is removed from the blockchain. If the contract can be killed by any arbitrary account, it is considered vulnerable [16]. It was this kind of vulnerability that was found and exploited in the updated Parity multisig wallet from the previous exploit. The vulnerability remained undetected from the previous fix for 3 months and 20 days until it was exploited.

Parity's multisig wallet is comprised of two distinct layers, in an "inheritance" like fashion: a light contract is deployed with every multisig wallet, that has minimal internal functionality and a library contract that is deployed only once, containing the main wallet logic. This division of responsibility makes the deployment of the light contract cheaper as all the logic in the library contract does not need to be re-deployed for each wallet but rather can reuse the library contracts logic [29].

In principle, this design paradigm should be followed when possible to make the deployment and utilization of contracts cheaper. The Parity developers, however, made a fundamental mistake: what they had deployed as a library was in fact an uninitialised wallet, enabling anyone to initialize it and claim ownership of the contract. This can be seen in the code snippit 2 through the `initMultiowned` function. Once initialized, the new owner of the library contract could then call the `kill` function, removing the library contract from the blockchain.

All of Parity's multisig wallets rely on this library for withdrawal functionality. Once the library was deleted, all existing wallets reliant on this library were rendered unavailable, with no way of accessing the funds stored within these wallets. This resulted in 514,000 Ether being locked up, unable to ever be moved, valued at 155 million USD at the time of the exploit [24].

The solution to this vulnerability is again very trivial: merely setting the `initMultiowned` function to `internal` would prevent this kind of exploit. This exploit is considered different to the incorrect contract modifier vulnerability as the attacker has to call two distinct functions to execute the exploit: one to set ownership of the contract and a second one to kill the contract [17].

Maian, a symbolic analysis tool for identifying vulnerabilities in smart contracts, is able to identify the suicidal contract vulnerability with a 99% positive rate [17]. In the Maian research, nearly one million contracts were analyzed and from this 1,495 contracts were identified, including the `ParityWalletLibrary` contract, vulnerable to this kind of attack.

The Maian research also identified "prodigal contracts", contracts that may leak Ether to arbitrary Ethereum addresses. From their research, they identified 1,504 contracts vulnerable to this exploit. They identified that the maximum amount of Ether that could be withdrawn from prodigal and suicidal contracts is nearly 4,950 Ether [17].

Additionally, the Maian tool can detect "greedy contracts", contracts that lock Ether away indefinitely, forever preventing its release. The Parity `enhanced-wallet` falls under this definition, with funds locked away forever. The Maian researchers identified 31,201 greedy contracts, making up 3.2% of the current contracts present on the blockchain [17].

V. Code Quality Audits with Static Analysis

Common vulnerabilities, along with tools capable of identifying them, have been discussed in the sections above. Next, a quantifiable way of defining code quality is outlined, through the use of static analysis. These static analysis tools would have been able to prevent some of the previously discussed hacks, primarily those relating to incorrect function modifiers definitions. Moreover, through static analysis, code consistency can be assured across a group of developers within a team, standardizing smart contract development practice. These analysis tools can be integrated into build systems, ensuring that only contracts that are free of known vulnerabilities and conform to predefined coding standards are deployed.

Three major open source Ethereum projects code bases (Aragon, Augur and OpenZeplin) are analysed using two of the most popular static analysis tools, Solium and Solhint [23], [30]–[33]. These repositories are compared over a number of releases, tracking the warnings and errors found in each of the releases. From this, clear trends can be identified to monitor the state of the code base.

A. Linter Comparison

The two most popular linters and code checkers in this space today are Solium and Solhint [22], [23]. Both provide the ability to check for stylistic problems and potential security vulnerabilities. The possible errors that the linters can identify have been outlined in Appendix 2. Solium is more widely adopted, used in a multitude of large open source projects. The three open source projects tested in this paper all had included Solium configuration files within their repositories, indicating that the development teams actively use these tools in production. Solhint claims to be faster and have more coding styling and security rules than Solium [33]. Both linters provide integration into common IDE's. Solium does not strictly adhere to the Solidity styling guide while Solhint does confirm to the Solidity Styling guide.

Solium detects a total of 32 distinct security violations, of which 13 are enabled by default [30]. It also detects 25 stylistic issues. Solhint, on the other hand, detects 18 distinct security violations, all enabled by default, with 22 stylistic errors. Solhint also detects 7 "best practice" rules [23].

The tests that follow involved testing using both Solium and Solhint, in an attempt to quantifiably compare the two. The public repositories for the open source projects were cloned and specific release numbers were checked out. Then, the linters were applied to code base at the point of release. The number of errors and warnings was recorded and plotted. These results can be seen in appendix 1.

B. *Analysis of Linter Output*

Solium and Solhint found different numbers of warnings and errors across all 3 repositories, over all releases. Clearly, this is due to the difference in the metrics used to quantify security and styling issues. In all cases, Solhint detected a larger total number of warnings and errors than Solium. The number of errors detected here is of less importance than the general trend of the graphs.

The general shape of the summation graphs across the three repositories is relatively consistent. All three show a steep drop in Solium warnings and errors during the release cycle, approximately halfway through the project's life. For example, Augur-Core experienced a drop from 484 warnings and errors to 36 between releases `0.12.0` and `0.12.0-1`. Despite this drop, the Solhint errors and warnings did not change considerably over these releases. This drop present in all repositories is due to the developers modifying the `.soliumrc.json` file between releases, as well as conducting code reformatting, resulting in a drop in the total number of errors and warnings that the linter flagged.

None of the repositories had any custom configuration for Solhint, meaning that this linter preformed all default executions, without ignoring any custom configurations.

AragonOS showed an interesting trend between releases `2.0.1` and `3.0.0-1` where the Solium total warnings and errors went down from 59 to 10, and the Solhint errors and warnings went up from 216 to 311. This change in direction is due to the developers defining their own styling rules before the second release, implementing it into the Solium configuration. This styling guide varied from what Solhint defined as correct and so an increased number of errors was observed here.

The take away from these graphs is not that one utility is better but rather that a linter is required to implement consistent styling and security within a team. Through the consistent use of a linter tool, developers can ensure consistency between releases, tracking against their own metrics and previous scores.

VI. Online Static Analysis Tools

A selection of online static analysis tools have been created, enabling a developer to upload smart contracts to check for common security vulnerabilities and formatting problems. The three most popular of these are SmartCheck, RemixIDE and Securify [20], [21], [34]. These tools do not analyze as many security and styling problems as the aforementioned symbolic execution tools or linters but do offer simplicity in utilization as they do not require any custom software to be installed to run checks. Moreover, the Remix IDE offers a browser based Solidity compiler, with the integrated ability to deploy contracts to a Javascript VM. Remix also offers a debugger, capable of analyzing the instruction set, all state variables, memory states and call stacks. This proves to be very useful when debugging contracts, a feature that none of the other tools offer. Lastly, Remix lets you utilize an injected Web3 provider or a local web3 provider, thus enabling you to deploy contracts using Metamask or a local Geth client (or other local provider, such as Parity) [35]–[37].

VII. Summary and comparison of Tools Used

A collection of tools have been used in this paper, all capable of detecting different errors and styling guides. Unfortunately, there is not a single solution to deal with all smart contract security and styling issues currently and so a combination is required to ensure adequate contract auditing.

A. *Symbolic Execution-Based Tools*

The two primary tools used in this paper were Maian and Oyente. Both tools provide useful insight into smart contract security, detecting a range of exploits and attacks. Due to the fact that each tool analyzes contracts in a different way and that they detect a different selection of errors, utilization of both tools is recommended to find all possible known problems within contracts.

Zeus is another symbolic execution tool, one that was not used in this paper as it's source code is not yet available online and so direct comparison of the tool is not possible [13].

B. *Code Linters*

Two popular code linters were compared, naimly Solium and Solhint. Both proved capable of detecting common styling and security errors. Solhint proved to find more errors than Solium, but this does not necessarily indicate the results are of higher quality. Both linters have IDE integration, enabling linting on the fly during the development process. One key difference identified between the tools is Solium's ability to be integrated into a build system. Solhint does not offer this functionality at the time of this paper and so recommendation must be made to use Solium. Additionally, the Solium command line interface (CLI) is easier to use as it can automatically recursively search directories for smart contracts while Solhint can't, requiring a more complex usage of regular expression within the CLI.

C. *Other Tools Worth Mentioning*

Mythril was not directly used in any tests during this paper as its functionality falls under a mixture of symbolic execution-based tools and linters [19]. It does provide a relatively extensive series of tests, including the infamous reentrancy vulnerability. It could prove to be a useful tool, used in conjunction with other tools but is not strictly needed if a symbolic execution tool as well as a linter is used.

Code coverage checkers, such as Solidity-Coverage, were also not used in this paper as these tools do not directly improve code quality or increase security but rather indirectly inform the developer of sections of code that require additional unit testing [38]. A code coverage checker integrated into CI can synergise well with an automated code linter. If a project has a lot of smart contracts and associated tests, a code coverage checker is recommended.

Online IDEs and code checkers, such as SmartCheck, RemixIDE and Securify all prove to be useful for quick security and styling audits. However, a proper linter and symbolic execution tool conduct more thorough testing in most cases and can be automated with CI so these kinds of code checkers are not recommended for larger projects. That being said, Remix IDE is arguably the most powerful Solidity IDE and can work in conjunction with a offline linter and symbolic execution tool through Remix's ability to interact with local files through the online IDE.

VIII. SMART CONTRACT DEVELOPMENT SECURITY BEST PRACTICES RECOMMENDATIONS

An ideal code testing suite, based on the currently available tools, can now be recommended.

A. *Symbolic Analysis*

A combination of Maian and Oyente to try and find all possible exploits is recommended.
Alternative: Mythril is capable of finding many errors but does not offer offline execution of contracts to verify true positives and its analysis is less thorough.

B. *Linter*

Solium for easy setup and configuration customization. Provides integration with CI, such as Jenkins.
Alternative: Solhint or online Linter.

C. *IDE*

Remix IDE for easy debugging and deploying, with built in Linter and basic static analysis.
Alternative: Sublime Text 3, Atom, VS code, Emacs or Vim with Solium Integration to provide in editor linting.

D. *Other Recommended Tools*

Code coverage checker, such as Solidity-Coverage.

Note that the above recommendations are development framework agnostic, able to work in conjunction with Truffle or Embark [19].

IX. CRITICAL EVALUATION OF TESTING SUIT

While the aforementioned combination of linters and code checkers is indeed capable of identifying a wide selection of problems within smart contracts, it clearly can not detect problems that are currently unknown to the development community. New exploits and hacks would most likely be missed by this testing suite, leaving the contracts vulnerable to being exploited. Despite this, the utilization of the proposed suit would ensure that past detected vulnerabilities and issues are identified quickly, ensuring security of contracts as far as possible.

The research into this field is still very new, it can even be considered bleeding edge. New tools are created everyday to analyze smart contracts for vulnerability and security problems and so keeping up to date with the latest developments in this space is vital to ensuring consistent security in smart contract development.

X. HOW TO RUN TOOLS DISCUSSED IN THIS PAPER

All tools discussed in this paper have extensive instructions on how to install and run them. The tool creator is best suited to providing these instructions. There are however complete instructions on how to run all tools discussed, available here on this papers Github Repository.

XI. CONCLUSION

Ethereum's technological context was presented, outlining the basics of blockchain and smart contracts. Three well known hacked contracts were discussed, along with solutions to the exploits and tools capable of identifying these exploits. Two popular Solidity linters were compared, using three popular open source code bases as a testing platform. Lastly, best practices in smart contracts development security were discussed, outlining a recommended development and testing suite.

REFERENCES

[1] Bitcoinwiki, “Proof of work - Bitcoin Wiki.” [Online]. Available: https://en.bitcoin.it/wiki/Proof_of_work

[2] Bitcoin.org, “51% Attack, Majority Hash Rate Attack - Bitcoin Glossary.” [Online]. Available: https://bitcoin.org/en/glossary/51-percent-attack

[3] S. Nakamoto, “Bitcoin: A Peer-to-Peer Electronic Cash System,” 2009. [Online]. Available: www.bitcoin.org

[4] Vatalic Buterin, “Ethereum White Paper,” 2018. [Online]. Available: https://github.com/ethereum/wiki/wiki/White-Paper

[5] V. B. Yoni Assia, “Colored Coins white paper - Digital Assets - Google Docs.” [Online]. Available: https://docs.google.com/document/d/1AnkP_cVZTCMLIzw4DvsW6M8Q2JC0lIzrTLuoWu2z1BE/edit

[6] BitcoinWiki, “Smart Property - Bitcoin Wiki,” 2016. [Online]. Available: https://en.bitcoin.it/wiki/Smart_Property

[7] Namecoin, “Namecoin,” 2015. [Online]. Available: https://namecoin.org/

[8] Nick Szabo, “The Idea of Smart Contracts.” [Online]. Available: http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/idea.html

[9] Alyssa Hertig, “What is Ethereum? - CoinDesk Guides,” 2016. [Online]. Available: https://www.coindesk.com/information/what-is-ethereum/

[10] S-tikhomirov, “smart-contract-languages Github,” 2018. [Online]. Available: https://github.com/s-tikhomirov/smart-contract-languages

[11] Solidity, “Solidity Solidity 0.4.23 documentation,” 2018. [Online]. Available: https://solidity.readthedocs.io/en/v0.4.23/

[12] Ameer Rosic, “What is Ethereum? A Step-by-Step Beginners Guide [Ultimate Guide],” 2016. [Online]. Available: https://blockgeeks.com/guides/ethereum/

[13] M. Dhawan, “Analyzing Safety of Smart Contracts,” 2018. [Online]. Available: https://isrdc.iitb.ac.in/blockchain/workshops/2017-iitb/papers/paper-12-AnalyzingSafetyofSmartContracts.pdf

[14] Thijs Maas, “Yes, this kid really just deleted $300 MILLION by messing around with Ethereum’s smart contracts.” 2017. [Online]. Available: http://tiny.cc/4rzdty

[15] Melonproject, “Oyente Github,” 2018. [Online]. Available: https://github.com/melonproject/oyente

[16] R. Baldoni, E. Coppa, D. Cono, D. ’elia, C. Demetrescu, and I. Finocchi, “A Survey of Symbolic Execution Techniques,” 2017. [Online]. Available: https://arxiv.org/pdf/1610.00502.pdf

[17] I. Nikoli, A. Kolluri, P. Saxena, and A. Hobor, “Finding The Greedy, Prodigal, and Suicidal Contracts at Scale,” 2018. [Online]. Available: https://arxiv.org/pdf/1802.06038.pdf

[18] Trailofbits, “Manticore Github,” 2017. [Online]. Available: https://github.com/trailofbits/manticore

[19] Consensys, “Truffle Suite - Your Ethereum Swiss Army Knife,” 2018. [Online]. Available: http://truffleframework.com/

[20] Securify, “Securify Formal Verification of Ethereum Smart Contracts,” 2018. [Online]. Available: https://securify.ch/

[21] Smartdec, “SmartCheck,” 2018. [Online]. Available: https://tool.smartdec.net/

[22] Duaraghav8, “Solium Security rules Github,” 2018. [Online]. Available: https://github.com/duaraghav8/solium-plugin-security/blob/master/README.md/#list-of-rules

[23] Protofire, “Solhint Github,” 2018. [Online]. Available: https://github.com/protofire/solhint

[24] Etherscan, “TheDAO wallet on the Etherscan,” 2016. [Online]. Available: https://etherscan.io/address/0xbb9bc244d798123fde783fcc1c72d3bb8c189413

[25] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, “Making Smart Contracts Smarter,” 2016. [Online]. Available: http://www.comp.nus.edu.sg/~loiluu/papers/oyente.pdf

[26] BlockCAT, “On the Parity Multi-Sig Wallet Attack BlockCAT Medium,” 2017. [Online]. Available: https://medium.com/blockcat/on-the-parity-multi-sig-wallet-attack-83fb5e7f4b8c

[27] Wolfie Zhao, “$30 Million: Ether Reported Stolen Due to Parity Wallet Breach - CoinDesk,” 2017. [Online]. Available: https://www.coindesk.com/30-million-ether-reported-stolen-parity-wallet-breach/

[28] Lorenz Breidenbach, Phil Daian, Ari Juels and E. G. Sirer, “An In-Depth Look at the Parity Multisig Bug,” 2017. [Online]. Available: http://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/

[29] Sergey Petrov, “Another Parity Wallet hack explained Sergey Petrov Medium,” 2017. [Online]. Available: https://medium.com/@Pr0Ger/another-parity-wallet-hack-explained-847ca46a2e1c

[30] Solium, “User Guide Solium 1.0.0 documentation.” [Online]. Available: http://solium.readthedocs.io/en/latest/user-guide.html?highlight=security

[31] Aragon, “AragonOS Github,” 2018. [Online]. Available: https://github.com/aragon/aragonOS

[32] Augor, “Augur Core Github,” 2018. [Online]. Available: https://github.com/AugurProject/augur-core

[33] OpenZeppelin, “Zeppelin Solidity Github,” 2018. [Online]. Available: https://github.com/OpenZeppelin/zeppelin-solidity

[34] E. foundation, “Remix - Solidity IDE,” 2018. [Online]. Available: http://remix.ethereum.org

[35] Metamask, “MetaMask,” 2018. [Online]. Available: https://metamask.io/

[36] Go-Ethereum Authors, “Go Ethereum Downloads,” 2018. [Online]. Available: https://geth.ethereum.org/downloads/

[37] Parity, “Parity,” 2018. [Online]. Available: https://www.parity.io/

[38] Sc-forks, “solidity-coverage Github,” 2018. [Online]. Available: https://github.com/sc-forks/solidity-coverage

Appendices

1 Static Analysis Test Results

Tables and figures below show the three open source repositories static analysis results with Solhint and Solium. The general graph trend is emphasized through the summation of errors and warnings graphs.

1.1 Augur-Core

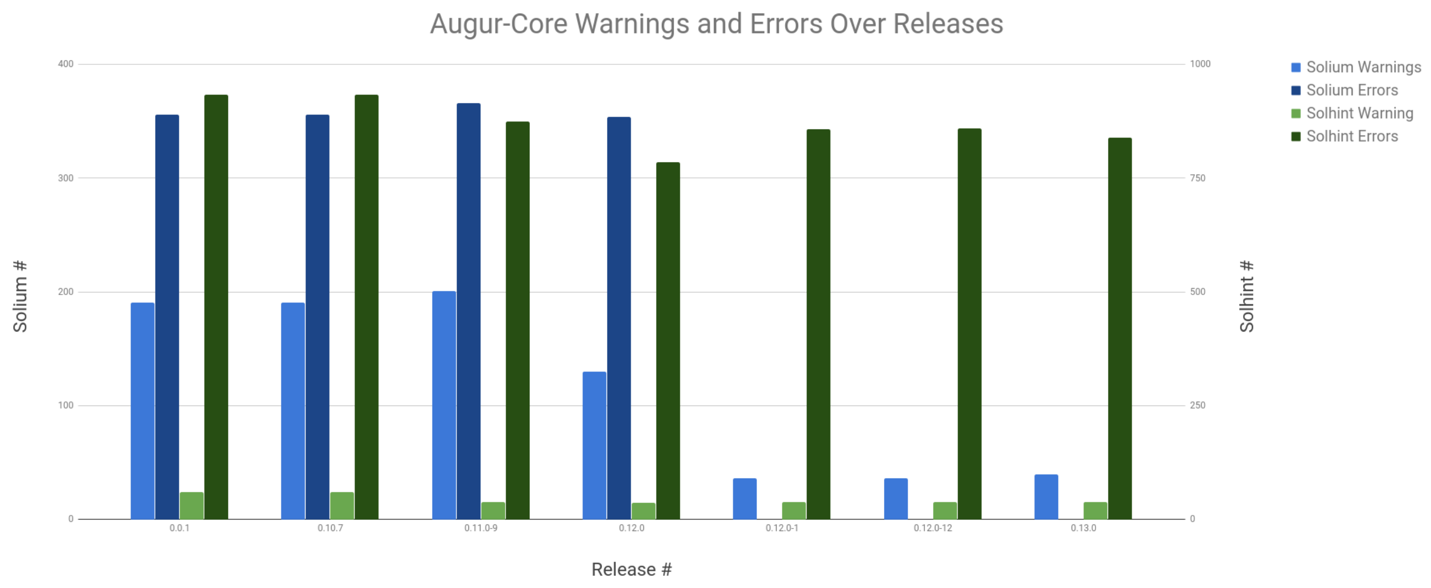


Figure 1: Normalized Augur-Core Test Results

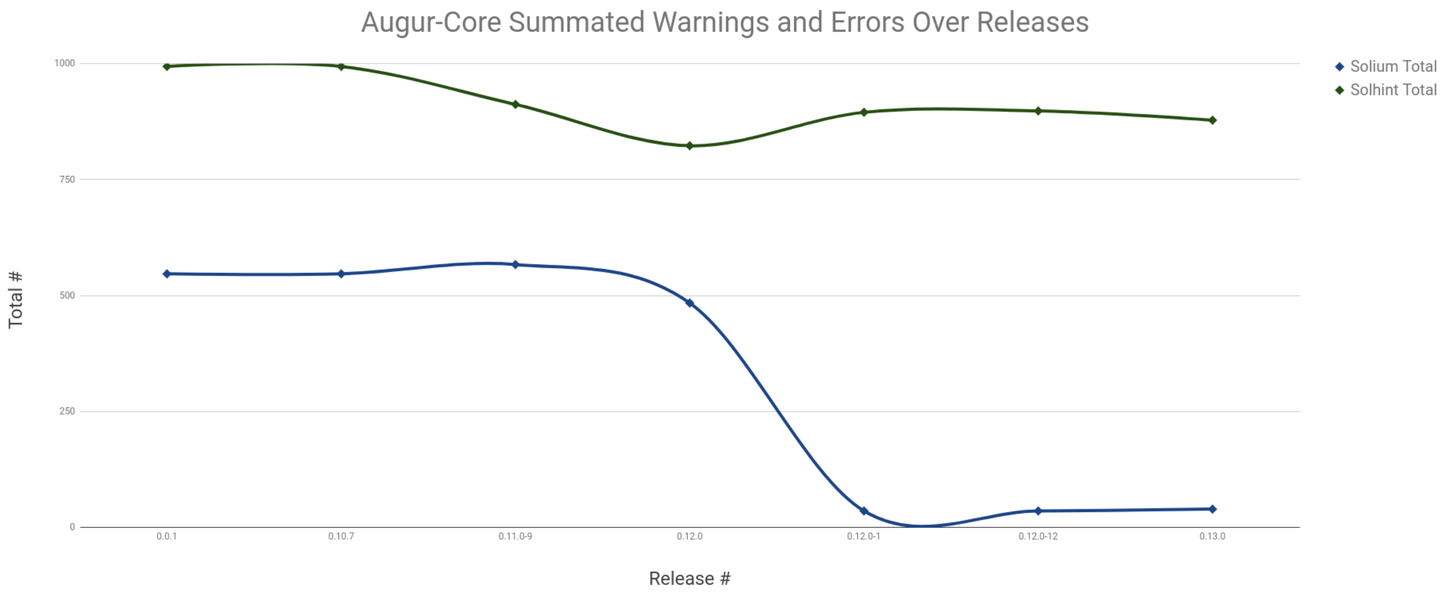


Figure 2: Summated Augur-Core Test Results

Table 1: Raw Augur-Core Test Results

Version	Release Hash	Solium Warnings	Solium Errors	Solium Total	Solhint Warning	Solhint Errors	Solhint Total
0.0.1	252562d	191	356	547	61	933	994
0.10.7	eda4e93	191	356	547	61	933	994
0.11.0-9	031f98c	201	366	567	38	874	912
0.12.0	f349ae1	130	354	484	37	786	823
0.12.0-1	a407972	36	0	36	38	857	895
0.12.0-12	92c02f4	36	0	36	38	860	898
0.13.0	9309329	40	0	40	38	840	878

1.2 AragonOS

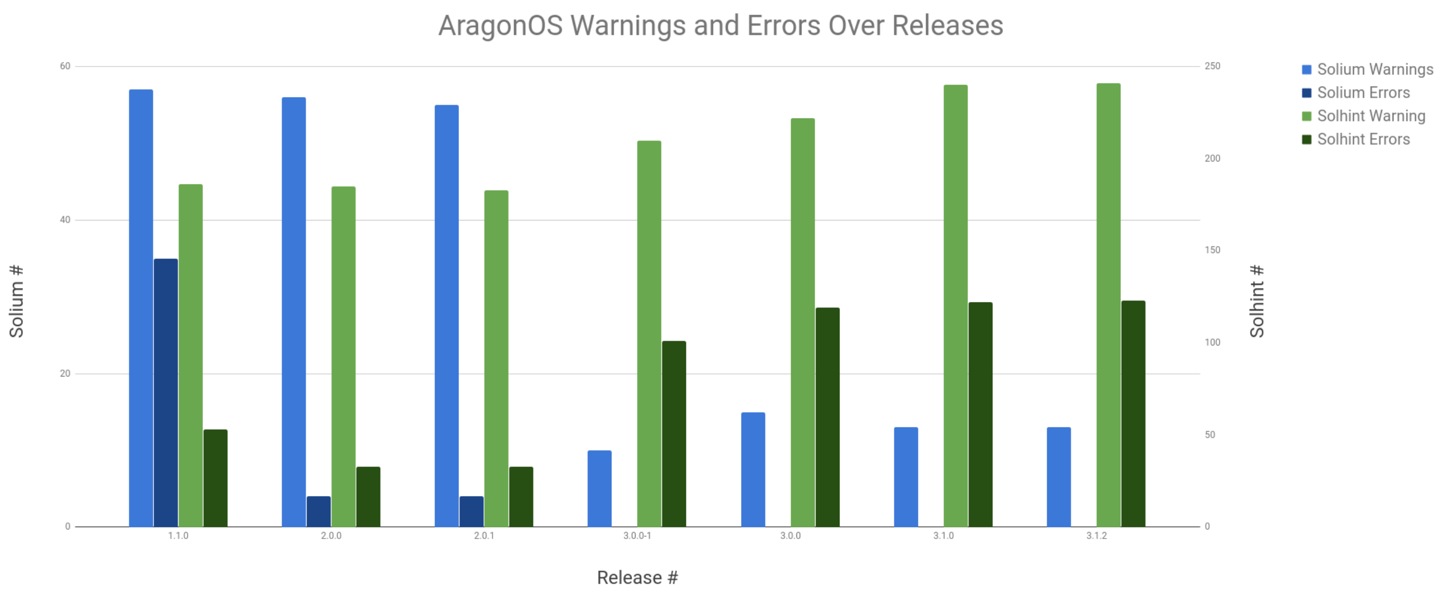


Figure 3: Normalized AragonOS Test Results

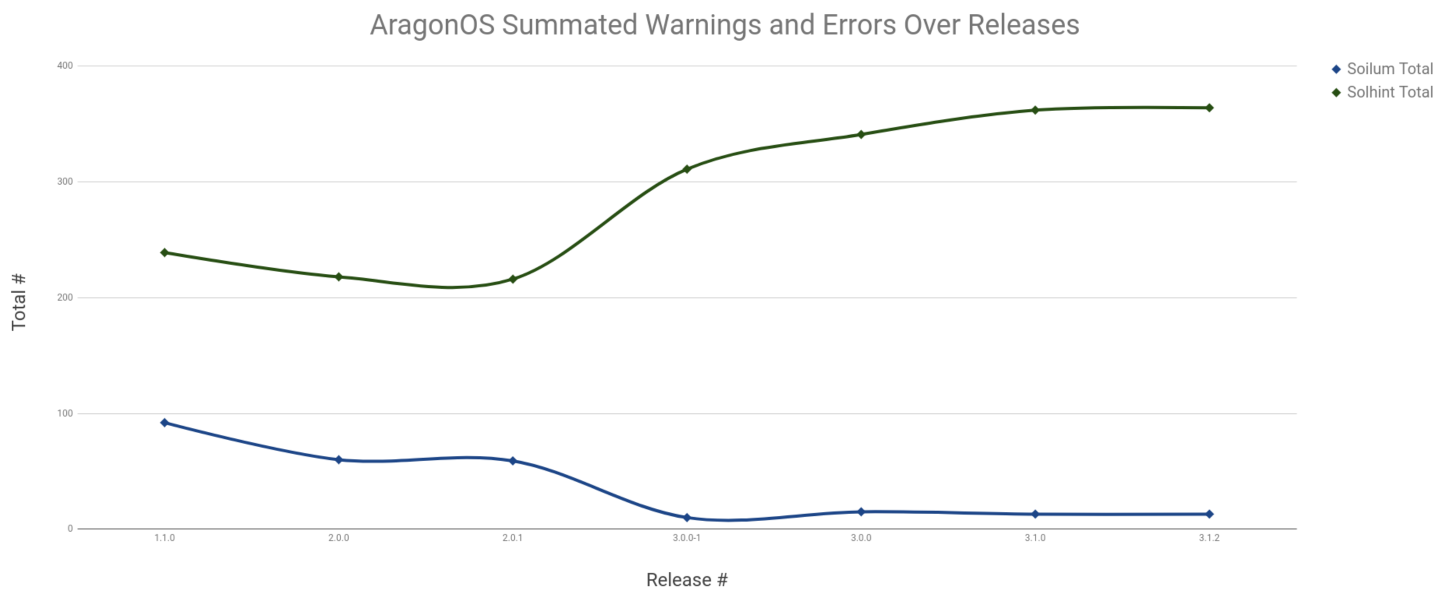


Figure 4: Summated AragonOS Test Results

Table 2: Raw AragonOS Test Results

Version	Release Hash	Solium Warnings	Solium Errors	Soilum Total	Solhint Warning	Solhint Errors	Solhint Total
1.1.0	fba8175	57	35	92	186	53	239
2.0.0	656593f	56	4	60	185	33	218
2.0.1	5816804	55	4	59	183	33	216
3.0.0-1	e6bbde7	10	0	10	210	101	311
3.0.0	f2e8b1a	15	0	15	222	119	341
3.1.0	501a905	13	0	13	240	122	362
3.1.2	4c03845	13	0	13	241	123	364

1.3 Solidity-Zeppelin

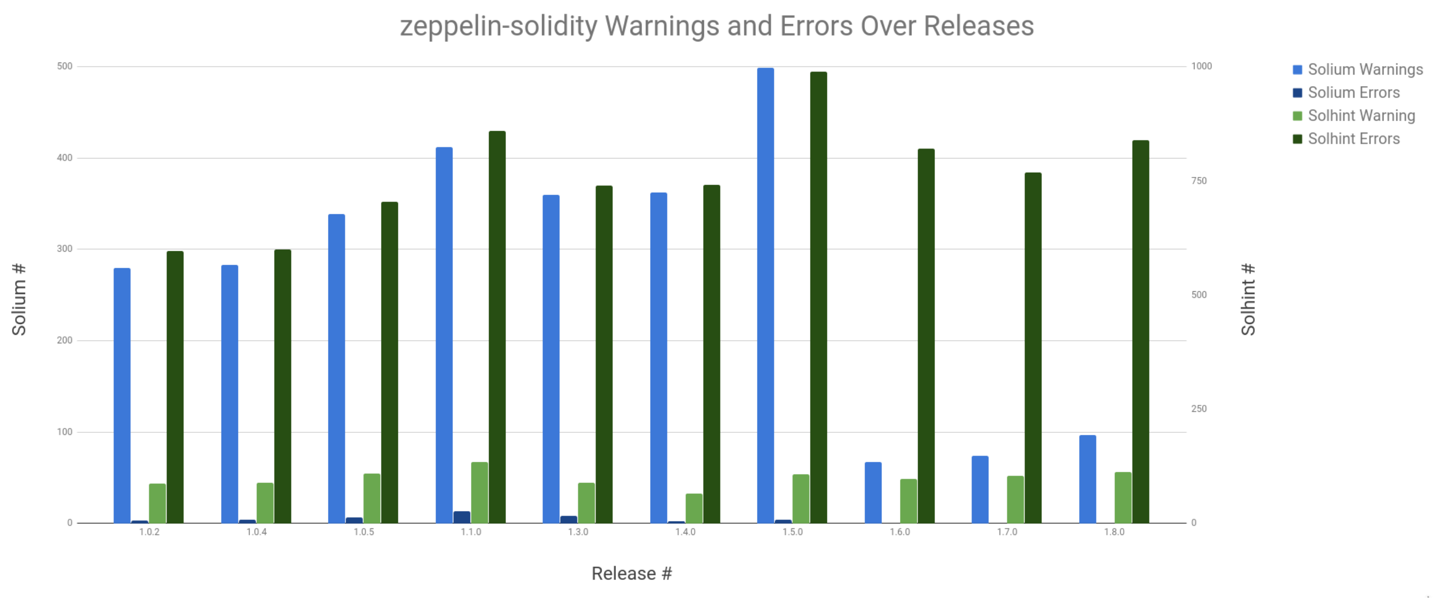


Figure 5: Normalized Solidity-Zeppelin Test Results

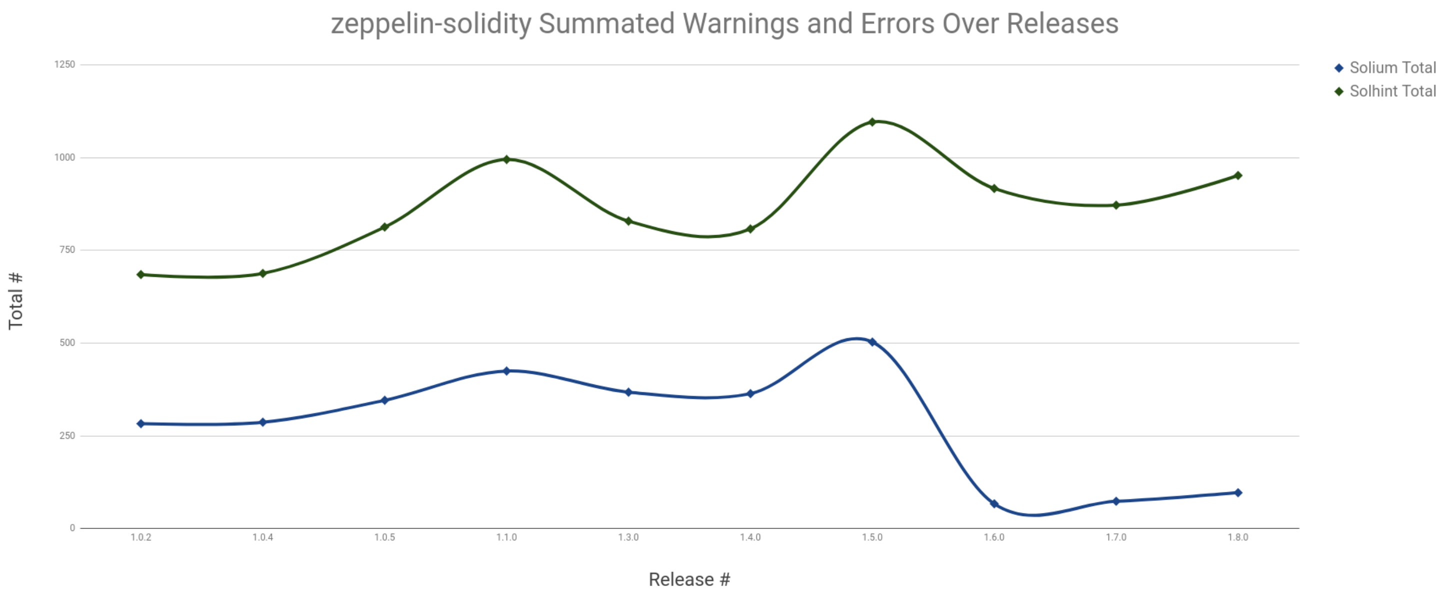


Figure 6: Summated Solidity-Zeppelin Test Results

Table 3: Raw Solidity-Zeppelin Test Results

Version	Release Hash	Solium Warnings	Solium Errors	Solium Total	Solhint Warning	Solhint Errors	Solhint Total
1.0.2	8cd5830	280	3	283	88	597	685
1.0.4	15169b2	283	4	287	89	599	688
1.0.5	7ac697b	339	7	346	109	704	813
1.1.0	7434b3d	412	13	425	135	860	995
1.3.0	725ed40	360	8	368	89	740	829
1.4.0	b7e7c76	362	2	364	66	742	808
1.5.0	4073cf6	499	4	503	107	989	1096
1.6.0	0541347	67	0	67	97	820	917
1.7.0	9ea4bae	74	0	74	104	768	872
1.8.0	cacf036	97	0	97	113	839	952

2 Linting Tools Error and Warning Detection Rules

A full list of all potential problems that the linting tools used (Solium and Solhint) can identify are shown in the five tables below.

2.1 Styling Rules

The tables below show the different styling rules the two tools can identify.

Table 4: Solium Styling Rules

Name	**Description**
imports-on-top	Ensure that all import statements are on top of the file
variable-declarations	Ensure that names 'l', 'O' & 'I' are not used for variables
array-declarations	Ensure that array declarations don't have space between the type and brackets
operator-whitespace	Ensure that operators are surrounded by a single space on either side
conditionals-whitespace	Ensure that there is exactly one space between conditional operators and parenthetic blocks
comma-whitespace	Ensure that there is no whitespace or comments between comma delimited elements and commas
semicolon-whitespace	Ensure that there is no whitespace or comments before semicolons
function-whitespace	Ensure function calls and declaration have (or don't have) whitespace in appropriate locations
lbrace	Ensure that every if, for, while and do statement is followed by an opening curly brace '{' on the same line
mixedcase	Ensure that all variable, function and parameter names follow the mixedCase naming convention
camelcase	Ensure that contract, library, modifier and struct names follow CamelCase notation
uppercase	Ensure that all constants (and only constants) contain only upper case letters and underscore
no-with [DEPRECATED]	Ensure no use of with statements in the code
no-empty-blocks	Ensure that no empty blocks {} exist
no-unused-vars	Flag all the variables that were declared but never used
double-quotes [DEPRECATED]	Ensure that string are quoted with double-quotes only. Deprecated and replaced by "quotes".
quotes	Ensure that all strings use only 1 style - either double quotes or single quotes
blank-lines	Ensure that there is exactly a 2-line gap between Contract and Funtion declarations
indentation	Ensure consistent indentation of 4 spaces per level
arg-overflow	In the case of 4+ elements in the same line require they are instead put on a single line each
whitespace	Specify where whitespace is suitable and where it isn't
deprecated-suicide	Suggest replacing deprecated 'suicide' for 'selfdestruct'
pragma-on-top	Ensure a) A PRAGMA directive exists and b) its on top of the file
function-order	Ensure order of functions in a contract: constructor,fallback,external,public,internal,private
emit	Ensure that emit statement is used to trigger a solidity event

Table 5: Solhint Styling Rules

Name	**Description**
func-name-mixedcase	Function name must be in camelCase
func-param-name-mixedcase	Function param name must be in mixedCase
var-name-mixedcase	Variable name must be in mixedCase
event-name-camelcase	Event name must be in CamelCase
const-name-snakecase	Constant name must be in capitalized SNAKE_CASE
modifier-name-mixedcase	Modifier name must be in mixedCase
contract-name-camelcase	Contract name must be in CamelCase
use-forbidden-name	Avoid to use letters 'I', 'l', 'O' as identifiers
visibility-modifier-order	Visibility modifier must be first in list of modifiers
imports-on-top	Import statements must be on top
two-lines-top-level-separator	Definition must be surrounded with two blank line indent
func-order	Function order is incorrect
quotes	Use double quotes for string literals
no-mix-tabs-and-spaces	Mixed tabs and spaces
indent	Indentation is incorrect
bracket-align	Open bracket must be on same line. It must be indented by other constructions by space
array-declaration-spaces	Array declaration must not contains spaces
separate-by-one-line-in-contract	Definitions inside contract / library must be separated by one line
expression-indent	Expression indentation is incorrect.
statement-indent	Statement indentation is incorrect.
space-after-comma	Comma must be separated from next element by space
no-spaces-before-semicolon	Semicolon must not have spaces before

Solhint also contains a selection of checks they define as "Best Practice Rules". These are shown in the table below.

Table 6: Solhint Best Practice Rules

Name	**Description**
max-line-length	Line length must be no more than 120 but current length is 121.
payable-fallback	When fallback is not payable you will not be able to receive ethers
no-empty-blocks	Code contains empty block
no-unused-vars	Variable "name" is unused
function-max-lines	Function body contains "count" lines but allowed no more than "maxLines" lines
code-complexity	Function has cyclomatic complexity "current" but allowed no more than "max"
max-states-count	Contract has "curCount" states declarations but allowed no more than "max"

2.2 Linting Security Rules

The tables below show the potential security vulnerabilities both tools can detect. OFF within the Solium table represents that the metric is turned off by default.

Table 7: Solium Security Rules

Name	**Description**
no-throw	Discourage use of 'throw' statement for error flagging
no-tx-origin	Discourage use of 'tx.origin' global variable
enforce-explicit-visibility	Encourage user to explicitly specify visibility of function
no-block-members	Discourage use of members 'blockhash' & 'timestamp' (and alias 'now') of 'block' global variable
no-call-value	Discourage use of .call.value()()
no-assign-params	Disallow assigning to function parameters
no-fixed	Disallow fixed point types
no-inline-assembly	Discourage use of inline assembly
no-low-level-calls	Discourage the use of low-level functions - call(), callcode() & delegatecall()
no-modify-for-iter-var	Discourage user to modify a for loop iteration counting variable in the loop body
no-send	Discourage the use of unsafe method 'send'
no-sha3	Encourage use of 'keccak256()' over 'sha3()' function
no-unreachable-code	Disallow unreachable code
OFF else-after-elseif	Encourage user to use else statement after else-if statement
OFF enforce-loop-bounds	Encourage use of loops with fixed bounds
OFF enforce-placeholder-last	Enforce that the function placeholder is the last statement in the modifier
OFF return-at-end	Discourage use of early returns in functions
OFF one-break-per-loop	Discourage use of multiple breaks in while/for/do loops
OFF max-statements-in-func	Enforce upper limit on number of statements inside a function
OFF no-abstract-func	Discourage use of abstract functions
OFF no-bit-operations	Disallow bitwise operations
OFF no-continue	Discourage use of 'continue' statement
OFF no-inheritance	Discourage use of inheritance
OFF no-multiple-inheritance	Discourage use of multiple inheritance
OFF no-named-params	Disallow named function parameters
OFF no-named-returns	Discourage use of named returns in functions
OFF 256-bit-ints-only	Disallow non-256 bit integers
OFF no-suicide-or-selfdestruct	Disallow suicide and selfdestruct
OFF no-var	Disallow type deduction via var
OFF no-user-defined-modifiers	Disallow user-defined modifiers
OFF no-void-returns	Discourage use of void returns in functions prototypes
OFF no-func-overriding	Discourage function overriding

Table 8: Solhint Security Rules

Name	**Description**
reentrancy	Possible reentrancy vulnerabilities. Avoid state changes after transfer.
avoid-sha3	Use "keccak256" instead of deprecated "sha3"
avoid-suicide	Use "selfdestruct" instead of deprecated "suicide"
avoid-throw	"throw" is deprecated, avoid to use it
func-visibility	Explicitly mark visibility in function
state-visibility	Explicitly mark visibility of state
check-send-result	Check result of "send" call
avoid-call-value	Avoid to use ".call.value()()"
compiler-fixed	Compiler version must be fixed
compiler-gt-0_4	Use at least '0.4' compiler version
no-complex-fallback	Fallback function must be simple
mark-callable-contracts	Explicitly mark all external contracts as trusted or untrusted
multiple-sends	Avoid multiple calls of "send" method in single transaction
no-simple-event-func-name	Event and function names must be different
avoid-tx-origin	Avoid to use tx.origin
no-inline-assembly	Avoid to use inline assembly. It is acceptable only in rare cases
not-rely-on-block-hash	Do not rely on "block.blockhash". Miners can influence its value.
avoid-low-level-calls	Avoid to use low level calls.