Software Assurance of Smart Contracts

Gabriel Bello   
*TSYS School of Computer Science*  
*Columbus State Univeristy*Columbus, GA, United States of America  
bello\_gabriel@columbusstate.edu

Gavin Kerr  
*TSYS School of Computer Science  
Columbus State Univeristy*Columbus, GA, United States of America  
kerr\_gavin@columbusstate.edu

line 1: 3rd Given Name Surname  
line 2: *dept. name of organization (of Affiliation)*  
line 3: *name of organization (of Affiliation)*line 4: City, Country  
line 5: email address

*Abstract*—Smart contracts are software found on blockchains, and they can be designed to execute automatically given a condition. Blockchain as a technology is designed for immutability, and as such, the code of a smart contract is intended to be unchangeable. Smart contracts have often been used to automate financial transactions on blockchains. Smart contracts, however, are susceptible to bugs, much like any other piece of code. As a result of the development process for contracts, which is often community-based and rooted in a lack of accountability, the code sometimes has bugs. Throughout the lifetime of blockchains and smart contracts, many bugs have been documented, some more famous than others. This paper documents many of smart contracts’ existing bugs with a focus on cross-platform bugs, succinctly listing the properties and effects of each. Subsequently, there is a categorization of the bugs based on CWE, which aids in determining the causes, effects, and properties of each bug.

Keywords—blockchain, smart contracts, bugs, software assurance, bug categorization, security.

# Introduction

Smart contracts provide a dynamic element to otherwise static blockchain systems that would require centralized oversight. A regular, paper contract dictates the legal agreement between two recognized parties on the ground of legal enforcement. A smart contract, on the other hand, is an automated piece of software that allows two or more parties to fulfill agreements without the need for federal enforcement. As smart contracts can potentially replace standard legal agreements in less intensive peer-to-peer conduct, their security is of great concern. With the primary purpose of blockchain systems is to manage and record transactions, the software behind automating these exchanges would need to be functionally secure.

Due to the autonomy of smart contracts, there is a necessity for adequate security mechanisms and practices to be in place to prevent exploitation. However, as it stands, smart contracts are not sufficiently regulated nor audited for existing vulnerabilities. Different platforms, languages, and implementations create roadblocks for widespread solutions. The goal of this project is to optimize the process of assurance in smart contracts by documenting known bugs and providing a baseline categorization by which smart contracts can be compared, analyzed, and improved.

Providing a bug reference for smart contract developers will contribute to the security of smart contracts as they are implemented. This project will also serve to provide context towards specific bugs; for example, there may be a bug present on a specific smart contract platform, but this project will document the vulnerability, state the underlying causes and effects, and potentially provide a solution. In this manner, the research project will contribute to the improved security of smart contracts such that there is opportunity for contracts to continue to grow. Other research in smart contract security provides solutions based on many different topics: protocol design, authentication methods, and many others. However, research to document and compile existing bugs continues to be a gap in the current scope of smart contract security. Our project seeks to work towards a solution.

The motivation for this research project revolves around providing a comprehensive list or reference for smart contract bugs. Smart contracts have a massive potential in many different industries, many of which necessitate compliance with privacy laws and standards. As a result, smart contract security is a purposeful avenue to follow, for it offers concrete contributions towards technological innovation. More and more, security is implemented throughout the development process of different products. However, smart contracts have known vulnerabilities and bugs that have not been eradicated on all platforms or contracts.

This work details many of the known bugs and vulnerabilities in smart contracts. The investigators focus on smart contract bugs that are cross-platform. To explain, this means that the focus is on bugs applicable to more than one platform that hosts smart contracts (e.g. a bug that could occur on Ethereum [1] and Tezos [2] blockchains). While compiling this list of cross-platform bugs, there is also a categorization of bugs by qualities, whether it is a cause, effect, or behavior of the bug. This project also documents the underlying practices or qualities that lead to a smart contract bug. By offering a baseline by which to standardize smart contract design from a coding perspective, smart contracts can be developed further while mitigating known vulnerabilities.

# Related Work

With blockchain’s fame in recent years, it is inarguable that much work has been done researching, designing, and implementing new platforms for different purposes. Whether this work is done by IEEE researchers or blockchain hobbyists, the outcomes are often valuable insight towards the overall standards of smart contracts. As more contracts are developed and more work is done, the community and knowledge surrounding software assurance grows stronger. Currently, there are many projects, analyses, and platforms that offer insights and solutions to smart contract bugs. However, the history of smart contracts provides just as much information toward the underlying design principles of smart contracts.

Nick Szabo introduces smart contracts as a “vending machine” like artifact that dispenses content based on a set of specific input condition [3]. The smart contracts overcome the simple functions of a vending machine having dynamic and proactive functionality [3]. The article also lists a set of three principles that should be followed in the construction of a secure smart contract [3]. While the list is rather compact, the principles abstracted to the point of being applicable to a diverse range of systems. Using this information, it is possible explore deeper into the infrastructures of different smart contracts.

Nick Szabo outlines a template for developing smart contract languages [4]. He describes particular focal points, such as contract negotiation and performance. Defined in the paper, we see a structure for how to operate these contracts (in pseudocode, as no language was in place in 2002) [4]. This information provides a fundamental understanding of the smart contract languages and could make apparent any flaws in the current architecture of smart contracts. This could also detail measures that need to be taken in relation to securing the operation of the smart contracts.

Hobor, Kumar, and Sergey provide an underlying analysis and defense of the programming language “Scilla” for the development of smart contracts [5]. While the bulk of this paper does not provide information on smart contract software assurance, the compiled list of smart contract languages for a variety of platforms is valuable [5]. Documentation of this nature is practically nonexistent and has great utility. This information helps with the organization of the analysis by providing a list of platforms on which to search for bugs.

# Scope and Methodology

When considering the different bugs that exist in smart contracts, it is most prudent to identify bugs that have the potential to affect multiple platforms. After analyzing the properties of documented bugs, a clear distinction can be made in what properties allow a bug to appear on across platforms. The bugs are then separated into “Platform” and “Cross-Platform” groups. “Platform” bugs can be understood to have originated from the underlying architecture of a given platform. Each platform has a unique architecture, which results in little cross-over from these bugs. However, there is also a possibility that one platform inherits the properties of another platform, which leads to shared architecture and cross-over of these bugs.

We choose, in this research, not to focus on platform-specific bugs, as their overall importance is less than cross-platform bugs, which may be found on multiple smart contract implementations. For example, the Ethereum Foundation runs and maintains the Ethereum Blockchain, which is one of the most popular public blockchains on the market [1]. The Solidity language, provided by the Ethereum Foundation for coding smart contracts on Ethereum, has required constant updating, as errors in the compiler have caused bugs with a varying severity [6]. The known errors have been collected in JSON format for public use on the Solidity website [6]. An example of a low-level bug from this collection would be the “OneOfTwoConstructorsSkipped” bug [6]. Because of a style shift in the Solidity language, constructors in different styles in the same code can cause the compiler to ignore correctly stated constructors [6]. The low-level nature of this bug can be attributed to how apparent the bug is in testing and styling conventions which deter developers from mixing programming styles.

This bug is specific to the Solidity language and are caused by an improperly designed core architecture. While information on these “Platform” bugs are of major importance to correlating developers, it is the platform manager’s main concern to document and repair these bugs. Contrariwise, “Cross-Platform” bugs may originate with the developers themselves. The developing principles for a smart contract on one platform can be transmitted to another, as they share common characteristics that cannot be entirely erased. With that in mind, documentation in regard to “Cross-Platform” bugs can both benefit current smart contract development but also prevent future platforms from hosting poorly implemented smart contracts. For these reasons, this work focuses on the collection and analysis of “Cross-Platform” bugs that have the potential to affect smart contracts as a whole.

The collection of bugs was mostly done through internet resources, such as blogs and online articles that outlined the adequate information. However, blockchain technology has been celebritized to no end, and the financial opportunities have attracted a large amount of amateur developers. Information about these bugs are framed in a much less technical sense which creates a lack of details that would otherwise be useful in finding a solution. Any information made available on mainstream outlets is filtered for the average consumer to digest. This means that an aspect of bias filtering must be included within the analysis portion that typically wouldn’t be required for such a topic.

For instance, in an article in The Merkle discussing the ShadowFork bug, the author uses biased language when discussing the business model used in the development of the smart contract [7]. Comments, where more technically correct language are appropriate, are replaced with the term ‘Ponzi scheme’ [7]. The inherent assumption of malice detracts from the discussion bug’s cause and available solutions. Buntinx gave little attention to the specifics of the bug, with most details generalized and seemingly quoted without cause or source [7]. Due to the constant flow of information and lag time in documenting that information in peer reviewed sources, information of this caliber must be used to some utility in place of more academic sources for most cases.

Once a diverse set of documented bugs had been compiled, each bug is analyzed and deconstructed to its underlying components. These components are compared to each other on the basis of cause and effect to determine common categorical flaws. The Common Weakness Enumeration (CWE) is utilized as a guiding tool for analyzing these components in addition to standardizing the descriptive syntax [8]. By making use of the CWE ids, external information about each bug is obtained through the CWE database [8]. This database also provides relational information that could be useful when applying this research to alternate domains.

# Bug Documentation and Categorization

The bugs listed in this section were retrieved from various sources. Each bug was documented, explained concisely, and categorized based on fundamental components, including cause, effect, and impact. After the collected bugs were listed and described, they were categorized using CWE Weakness Types [8]. Each bug has its components analyzed and deconstructed until they could be generally described in groups based on common traits. These traits were then linked to appropriate Weakness Types, and bugs were grouped under one or more identifiers. Listed here will be the Weakness Types with the linked bugs grouped underneath.

## Categories of CWE

**CWE-248: Uncaught Exception**

**Description**: An exception is thrown from a function, but it is not caught.

**CWE-269: Improper Privilege Management**

**Description**: An actor is given unintended privileges

**CWE-362: 'Race Condition'**

**Description**: Concurrent code sequences that share resources but do not release resources properly.

**CWE-367: Time-of-check Time-of-use (TOCTOU) Race Condition**

**Description**: Inconsistency in the state of a resource between the time of check and use.

**CWE-460: Improper Cleanup on Thrown Exception**

**Description**: Unexpected state from improper cleanup of exception.

**CWE-665: Improper Initialization**

**Description**: Resource is not correctly initialized

**CWE-696: Incorrect Behavior Order**

**Description**: Weakness that results from behaviors that execute in the wrong order.

## CWE-248: Uncaught Exception

**Mishandled Exception Bug**

The Mishandled Exception Bug, results from improper propagation of ‘callee’ exceptions to the ‘caller’ [9]. Given that one smart contract can ‘call another,’ the improper handling of these exceptions can lead to unexpected behaviors in the caller that can detriment both parties [9]. In an accidental instance, users would be unable to make use of functionality that would have been otherwise available while also having to deal with unintended side effects. Malicious user can intentionally throw exceptions to make the caller act in specific manner that suits their needs.

**Unchecked Send Bug**

The Unchecked Send bug is possible when the send function fails, which can happen for multiple reasons, and the user doesn’t get their currency delivered [10]. Essentially, this bug is a same as the Mishandled Exception Bug but is specific to the send function that is core to the smart contract functionality [9]. However, because of the poor handling of the send function by the developers, this is categorized as a “Cross-Platform” bug. Due to the prevalence of the send function being used in smart contracts as a whole, the Unchecked Send Bug [10] has the potential to affect a given smart contract more than the Mishandled Exception Bug [9]. As a result, the two should be considered separately from a pragmatic standpoint in order to specify potential outcome.

## CWE-269: Improper Privilege Management/ CWE-665: Improper Initialization

**Parity Bug**

The Parity bug resulted from a situation in which the owner of a smart contract was not initialized when the smart contract was first deployed [11]. By accident, another individual was able to initialize himself/herself as the owner and executed the kill function [11]. Once the kill function is executed, a smart contract is disabled along with other mechanisms being executed to handle the associated tokens. If these other mechanisms are not set in place by the developers, then the associated tokens will become inaccessible. This specific bug originated in the deployment stage of the development as not all the necessary information was initialized upon activation. A malicious user would have free reign of the smart contract if he/she was able to assign themselves the role of owner. While the effects of this bug were associated primarily with the architecture of the Ethereum [1] platform, the actual cause of the bug could be found any platform as it resulted from poor initialization.

## CWE-362: 'Race Condition'

**Transaction Ordering Dependence**

Luu et al. detail a set of bugs on the Ethereum platform [9]. The first, called Transaction Ordering Dependence (TOD), occurs due to uncertainty in the state of a given smart contract due to ‘individual invocations’ and a ‘dependence on the transaction ordering’ [9]. [9] notes that given that ‘’ is a state of a blockchain and that ‘’ and ‘’ are transaction on a new block in addition to invoking the same contract, the following example applies:

is applied when the contract is at either state or state where , depending on the order between and . Thus, there is a discrepancy between the state of the contract that users may intend to invoke at, and the actual state when their corresponding execution happens. Only the miner who mines the block can decide the order of these transactions, consequently the order of updates. As a result, the final state of a contract depends on how the miner orders the transactions invoking it [].

This can be agitated accidentally or invoked by a malicious user. Without being able to say for certain the state of the smart contract at a given time, concurrent actions can have a major impact on the result.

## CWE-367: Time-of-check Time-of-use (TOCTOU) Race Condition

**Timestamp Dependence Bug**

The Timestamp Dependence Bug stems from smart contracts that utilize timestamps to trigger conditions [9]. Normally, there is some room for error when timestamping the blocks, allowing for a malicious user to partially control the state of the contract [9]. Depending on timestamp’s level of involvement on the functionality of the smart contract’s operation, the malicious user can act with varying levels of harm.

## CWE-460: Improper Cleanup on Thrown Exception

**ShadowFork Bug**

The ShadowFork Bug was caused by an internal exception when Ether was to be withdrawn [7]. Any users that attempted to ‘remove funds’ would activate this exception and receive no return [7]. This is slightly different than the Mishandled Exception Bug [7] as the exceptions originates internally and affects directly the operation of the smart contract without the need for another contract.

## CWE-696: Incorrect Behavior Order

**DAO Bug**

The DAO Bug is unique in that the attackers were able to take advantage of the same vulnerability in two different fashions [12]. When the first version was resolved, the underlying cause was still present in the code allowing for a second attack. The bug is generally composed of recursive calls to the same function before execution can end or withdrawn amounts have been updated. The first attack that took advantage of this bug was the ‘Reentrancy’ attack. A short example of this is provided by [13]:

1. Something

// INSECURE

mapping (address => uint) private userBalances;

function withdrawBalance() public {

uint amountToWithdraw = userBalances[msg.sender];

require(msg.sender.call.value(

amountToWithdraw)()); // At this point, the caller's code is executed, and can call withdrawBalance again

userBalances[msg.sender] = 0;

}

By taking advantage of how the code in the withdraw function is ordered, a malicious user can reenter the function before the function has completed running. This allows them to withdraw the same amount without affecting their actual balance. Only once the malicious user has opted to not renter the function will all the previously invoked instances of the function be able to finish. Up until that point there is nothing stopping the malicious user from taking as many tokens as he/she desires.

The second attack is called the ‘Cross-function Race Condition’ and takes advantage of the same basic venerability [13]. Instead of making use of a single function, the attack can use two functions in the same smart contract that have the same state. Another example is provided by [13]:

1. Text

// INSECURE

mapping (address => uint) private userBalances;

function transfer(address to, uint amount)

{

if (userBalances[msg.sender] >= amount) {

userBalances[to] += amount;

userBalances[msg.sender] -= amount;

}

}

function withdrawBalance() public

{

uint amountToWithdraw = userBalances[msg.sender];

require(msg.sender.call.value(

amountToWithdraw)()); // At this point, the caller's code is executed, and can call transfer()

userBalances[msg.sender] = 0;

}

In Fig 2., the attacker can transfer out their tokens before the withdraw has finished executing. Because their balance hasn’t been updated properly, it is technically and logically valid for them to transfer their tokens. Afterward, the withdraw function finishes and the tokens are sent to the user. This doubles the number of tokens that a user can receive from the smart contract.

# Current assurance techniques in smart contracts

While smart contracts are a relatively new technology, as they were discussed a decade ago but only recently implemented, there are a few assurance techniques that developers can take advantage of to better prepare their smart contracts. With blockchain being an open marketplace, there is an incentive on the developer’s part to ensure safe and functional code to potential consumers. Currently there are three techniques available to developers: audits/code reviews and formal verification. No single technique can ensure that code is completely secure but using such techniques can reduce the number of bugs present in code.

Zeppelin Solutions provide a smart contract specific auditing service, with many of the audit reports made publicly available [14]. These reports go into detail about different flaws in the contracts according to severity [14]. Recommendations are recorded to guide the developers to an efficient and effective solution the flaws [14]. Additionally, the reports note whether the contracts were modified in accordance with the audit report recommendations [14].

Securify provides an automated system for formal verification of smart contracts [15]. Formal verification under normal circumstances can take long periods of time, but the automation aspect of this service shortens that period considerably. The automation also makes the service available to wider base than otherwise [15]. Any individual or group can use the system, with limited human interaction, to quickly verify their smart contract code [15]. ‘Extensibility’ is also listed as major quality of this service, allowing for the ‘capture [of] any newly discovered security vulnerabilities’ [15].

These are, however, not the extend of the services that are available, but just examples of assurance providers on the market. When used in conjunction, auditing and formal verification produce more secure code than unassured smart contracts. Considering that it is near impossible to fix a flawed smart contract once it is on the blockchain, it is of great importance that smart contracts are assured to the fullest extent.

# Roadblocks to increased assurance

Smart contracts are a new, innovative technology; as such, they are constantly evolving along with the research surrounding them. Many issues arise when developing a project around new technology, and this project has seen roadblocks relevant to smart contracts. We have worked to mitigate these issues by reworking our scope and focus. By building on the shoulders of previous work, this project can provide concrete additions to smart contract security.

One challenge for this project involves the varying platforms for smart contracts. Different implementations of smart contracts can result in varying levels of security due to several discrepancies between platforms. One example is varying languages for smart contracts. Even within the same platform, such as Ethereum, there can be multiple languages used to develop smart contracts. There are also fundamental differences in implementations and behaviors for smart contracts on platforms. Consequently, bugs may behave differently or be nonexistent on separate platforms.

A second complication that has come up in this project is the lack of substantial, organized, comprehensive research on smart contracts. There is a shortcoming in enveloping research, and much of the research done is community-based. The results can be questionable, even if they are valid the majority of the time. Additionally, there is not a wealth of technically-based research around bugs. Many of the projects revolve around protocol analysis and creation, which can overlap with this project - but it also may not. The lack of research could be seen as a benefit or a drawback to this project. On one hand, there is room for growth and innovation in unchartered territory. Contrarily, the process of developing such research is more difficult.

Categorizing the smart contracts proved to be an arduous task, especially considering the platform used. With over six-hundred different categories, the Common Weaknesses Enumeration (CWE) makes it difficult to succinctly categorize the smart contract bugs [8]. A more reasonable set of categorical types would prove more purposeful for discerning unique bugs. However, no publicly available resource exists in its complete form; even so, the challenge of categorization was not insurmountable.

# Future work

Smart contracts are growing in popularity, whether in research, amateur implementation, or business adoption. Consequently, there are several avenues to pursue in future work. Both at low and high levels, in coding and in auditing, smart contract research is in its infancy.

With this work in mind, there is room for more development in multiple areas. As more bugs are documented, there is a need to categorize them for reference – a never-ending task. Moreover, organizations that contribute to developing new platforms may build on the research done here; using this compilation of bugs as a reference to secure smart contract platforms comes with more auditing challenges. Work can be done here to bridge the gap between bug documentation and enacting software assurance in a platform.

The aforementioned CWE categorization method is useful and robust, but lacks the brevity for simplistic uses, such as this work [8]. A condensed list of categories would provide a more accurate baseline by which to measure smart contract programs (or all programs, for that matter).

# References

1. Ethereum Foundation (2018). *Ethereum Project*. [online] Ethereum.org. Available at: https://www.ethereum.org/ [Accessed 2 May 2018].
2. Tezos (2018). *Tezos*. [online] Tezos.com. Available at: https://tezos.com/ [Accessed 2 May 2018].
3. Szabo, N. (1997). *Nick Szabo -- The Idea of Smart Contracts*. [online] Fon.hum.uva.nl. Available at: http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/idea.html [Accessed 13 Apr. 2018].K. Elissa, “Title of paper if known,” unpublished.
4. Szabo, N. (2002). *A Formal Language for Analyzing Contracts*. [online] Fon.hum.uva.nl. Available at: http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/contractlanguage.html [Accessed 13 Apr. 2018].
5. Sergey, I., Kumar, A. and Hobor, A. (2018). *Scilla: a Smart Contract Intermediate-Level LAnguage*. [online] Arxiv.org. Available at: https://arxiv.org/pdf/1801.00687.pdf [Accessed 13 Apr. 2018].
6. Ethereum (2017). *List of Known Bugs — Solidity 0.4.22 documentation*. [online] Solidity.readthedocs.io. Available at: http://solidity.readthedocs.io/en/latest/bugs.html [Accessed 13 Apr. 2018].
7. Buntinx, J. (2018). *ShadowFork’s Smart Contract Bug Freezes $1M Worth of Ethereum*. [online] The Merkle. Available at: https://themerkle.com/shadowforks-smart-contract-bug-freezes-1m-worth-of-ethereum/ [Accessed 13 Apr. 2018].
8. text
9. Delmolino, K., Arnett, M., Kosba, A., Miller, A. and Shi, E. (2016). Step by Step Towards Creating a Safe Smart Contract: Lessons and Insights from a Cryptocurrency Lab. *Financial Cryptography and Data Security*, pp.79-94.
10. Miller, A. and Wen, Z. (2016). *Scanning Live Ethereum Contracts for the &quot;Unchecked-Send&quot; Bug*. [online] Hacking Distributed. Available at: http://hackingdistributed.com/2016/06/16/scanning-live-ethereum-contracts-for-bugs/ [Accessed 13 Apr. 2018].
11. Suiche, M. (2017). *The $280M Ethereum’s Parity bug. – Comae Technologies*. [online] Comae Technologies. Available at: https://blog.comae.io/the-280m-ethereums-bug-f28e5de43513 [Accessed 13 Apr. 2018].
12. Daian, P. (2018). *Analysis of the DAO exploit*. [online] Hacking Distributed. Available at: http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/ [Accessed 2 May 2018].
13. Gleim, B. (2016). *Ethereum Smart Contract Best Practices*. [online] Consensys.github.io. Available at: https://consensys.github.io/smart-contract-best-practices/ [Accessed 13 Apr. 2018].
14. Zeppelin Solutions (2018). *Security – Zeppelin Blog*. [online] Zeppelin Blog. Available at: https://blog.zeppelin.solutions/tagged/security [Accessed 13 Apr. 2018].
15. Software Reliability Lab (2018). *Securify ♦ Formal Verification of Ethereum Smart Contracts*. [online] Securify.ch. Available at: https://securify.ch/ [Accessed 2 May 2018].