The Risks and Challenges of Implementing Ethereum Smart Contracts

Christopher G. Harris
School of Mathematical Sciences
University of Northern Colorado
Greeley, CO 80639 USA
christopher.harris@unco.edu

***Abstract*—Smart contracts are designed to facilitate the performance of trackable and irreversible transactions without the need for third party involvement. Therefore, as a result of this lack of oversight, it is essential that these smart contracts are written and properly tested. In this paper, we examine some of the prominent risks and challenges involved with writing and implementing smart contracts and discuss how each of these challenges can be overcome. We focus on contracts executed on Ethereum, the most prominent smart contract platform.**

Keywords—Ethereum, smart contracts, Solidity, blockchain

I. INTRODUCTION

Smart contracts – one of the most widely-discussed applications to result from the rise of blockchain technology – permit companies to create trustless, automated agreements. These smart contracts are considered fulfilled only when specific conditions built into the contract are met. Not only do they facilitate the development of applications directly on the chain, but they also work with tokens that provide considerable utility for various types of financial transactions. These factors have facilitated considerable speculation over its potential to affect numerous industries from law to logistics to finance.

As a new technology smart contracts are not without numerous risks and challenges, many of which are a direct result of their stated benefits, such as lack of third-party oversight and their immutability. Therefore, they have drawn considerable criticism from many who have observed the resulting chaos. The most prominent framework for smart contracts is Ethereum [1], whose capitalization has exceeded $12.2B since its mid-2015 launch[1]. In Ethereum, smart contracts are rendered as computer programs written in Solidity, a Turing-complete language. In this paper, we examine these risks and challenges primarily in the context of Ethereum; however, many of these same issues can be seen on the 40+ other platforms used for smart contract development (for a comprehensive list, see [2]). Ethereum maintains a shared view of the global state using a proof-of-work consensus mechanism like that found in Bitcoin.

The remainder of this paper is organized as follows. In the next section, we discuss the essential properties of smart contracts. In Section III, we discuss some of the common types of bugs encountered in Ethereum and describe some of the approaches that have been used to address these bugs. Last, in Section IV, we discuss the general issues of risk in smart contracts and examine overall directions in their resolution.

[1] https://coinmarketcap.com/currencies/ethereum/

II. SMART CONTRACTS

In 1994, Nick Szabo introduced the concept of a smart contract. He defined it as "a computerized transaction protocol that executes the terms of a contract" [3]. Whereas a contract is an agreement between two or more parties that binds them to some future condition or state, according to Szabo smart contracts translate these contractual clauses (such as collateral and bonding) into the code and embeds them into hardware or software. This structure makes them self-enforceable in order to minimize the need for trusted intermediaries between transacting parties [4]. Blockchain technology, which provides the essential characteristics of immutability, irreversibility, decentralization, and persistence, have made smart contracts achievable [5].

A. Essential Properties of Smart Contracts

Smart contract platforms need to provide three properties: they need to be deterministic, isolated, and terminable.

The first property is **deterministic** – they need to produce the same result each time they run. Several conditions can affect deterministic behavior in a program:

- If it relies on an *external state* or relies on *dynamic or non-deterministic function calls*, such as those that depend on hardware timer values, random values or values that change over time.
- If it operates in a way that is *sensitive to timing or run order*, for example, if multiple processors are writing to the same data at the same time; the precise order in which each processor writes its data affects the result.
- If a hardware error causes the *program's state to change unexpectedly*.

Isolation is the second condition for a smart contract. A smart contract can be uploaded by anyone, which introduces a risk since any single contract may (by design or by accident) contain viruses and bugs. If the contract is not isolated, this could impact the entire blockchain ecosystem. Therefore, contracts need to be isolated in a sandbox to save the entire

978-1-7281-1328-9/19/$31.00 ©2019 IEEE

ecosystem from any negative effects that one contract could introduce.

The third condition, **termination**, is also essential in smart contracts since by definition a smart contract must be capable of completing within a specified time limit. There are three primary methods to guarantee termination in smart contract programs.

- *Turing Incompleteness*: To avoid entering an endless loop, a Turing Incomplete blockchain will have limited functionality and may not be capable of making jumps and loops.
- *Steps and Fee Meters*: A program can keep track of the number "steps" it has taken and then terminate once a step count has been reached. With a fee meter, contracts are executed with a pre-paid amount put into a reserve. Every instruction execution requires a specified fee. If the fee spent exceeds the pre-paid allocated amount, the contract is subsequently terminated.
- *Timers*: Here a pre-determined timer is maintained; if the contract execution exceeds the time limit then it is externally aborted.

Ethereum contracts have no non-deterministic functions and the available data is limited to on-chain information only. Dynamic calls can be executed, and these calls can be non-deterministic. Ethereum uses the fee meter approach for termination, requiring "gas" (a fee) to deploy and execute smart contracts; once the gas used exceeds the pre-paid allocated amount, the contract is terminated. The Ethereum Virtual Machine (EVM) language supports an instruction set of 150 8-bit opcodes (at the time of this writing, see [6] for a list). This instruction set allows the creation of Turing-complete programs that support smart contract execution and are thus able to express arbitrarily complex logic [7]; however, the number of instructions available in EVM is limited to ensure termination; otherwise, the miners could encounter endless loops, and formal blockchain verification would become impossible. For this reason, EVM is considered a quasi-Turing complete language in practice [9]. Ethereum provides a virtual machine environment assuring isolation but competing platforms such as Hyperledger Fabric use a docker, which is namespace dependent and thus cannot guarantee isolation.

III. Etherium-Based Bugs

We can divide these bugs into four categories: those related to callbacks, those related to integer errors, those related to language and version inconsistencies, and those related to execution inconsistencies.

A. Bugs related to callbacks

Unlike most standard programming environments like Java and C, Ethereum smart contracts lack a global mutable shared state. Ethereum blockchains (as well as numerous other dynamic environments) implement event-driven programming using callbacks. These callbacks allow programmers to break modularity – while they are essential for good programming style and code extendibility, they also have the adverse consequence of potentially compromising security.

Modularity is essential to blockchain development since contracts are often contributed by multiple sources, some of which may have malicious intent. A widely-reported bug in *The DAO* contract in June 2016 exploited callbacks to steal $150M [7], but there are many other bugs related to callbacks that did not get as much attention. A callback bug in *The DAO* allowed an adversarial contract to mutate *The DAO*'s state by calling back to it recursively. A function called *splitDAO()* was vulnerable to the recursive send pattern in which updates to user balances and totals were done *after* the movement of funds; the call the function to execute a split were allowed to occur before the initial withdrawal completed. This exploit was only rectified through the one-time use of a hard fork, thus violating the immutability principle of the blockchain.

As with many callback bugs, *The DAO* exploit was not due to any issues with the Ethereum opcodes, but due to poor programming and testing. As a result, several researchers have published methods to examine the use of reentrant code [10] or provide a static analysis of smart contracts [11]. As these methods mature, we anticipate checks of these types to be incorporated into the EVM bytecode before a contract can be implemented.

B. Bugs related to integer errors

There are many scenarios involving integer operations that can result in bugs in Ethereum smart contracts. Integer errors in smart contracts can be broken into three main categories: arithmetic bugs, truncation bugs and signedness bugs [9].

Arithmetic bugs include overflows (i.e., through opcodes ADD and MUL), underflows (i.e., through the SUB opcode), and division by zero or modulo zero. These can occur if the bounds conditions are not fulfilled by the resulting type. This results in a value that wraps around. All operations in EVM are modulo 2^{256} due to the 256-bit word size,, but if the addition of two unsigned numbers is greater than $2^{32}-1$ in Solidity (currently the primary language used in coding Ethereum contracts), it will wrap around, without producing an exception as would occur in other languages like Java or C. Likewise, in Solidity versions 0.4.0 and lower, division by zero or a number modulo zero will also produce a 0 and not an exception [12]. In April 2018, abnormal fluctuations were observed in BEC tokens. Attackers had successfully got 10^{58} BECs exploiting a vulnerability caused by an integer overflow [13]. The authors of [9] found that out of the 50,535 distinct contracts (i.e., non-copied versions) they investigated in 2018, 20,520 overflows were found in 21% of the contracts, 6,103 underflows in 12% of the contracts and 29 modulo division errors in less than 1% of contracts.

Truncation bugs occur when the precision of a value is lost, usually due to a conversion from one integral type to a narrower integral type. The speed of light in a vacuum is 299,792,458 meters per second. In notation, this quantity is expressed as 2.99792458×10^8. Truncating it to two decimal places yields 2.99×10^8. The truncation error is the difference between the actual value and the truncated value, or 0.00792458×10^8. Expressed properly in scientific notation, it

is 7.92458 x 10^5, which is a rounding error of approximately 0.26%. In computing applications, a truncation error is a discrepancy that arises from executing a finite number of steps to approximate an infinite process. In a financial transaction, this type of truncation error can result in the loss of Ether.

Signedness bugs occur when a conversion is made from a signed integer type to an unsigned integer type of the same width (or vice versa). This type of behavior can occur because internally to the computer, there is no distinction between the way signed and unsigned variables are stored. This class of bug can be problematic to exploit, since signed integers, when interpreted as unsigned, tend to be very large. If in Figure 1, the values supplied to a and b are both `0x7fffffff`, the bounds check in line 2 passes, and the resulting value is `0xfffffffe`, or -2, which is far less than the expected value.

```
1   function addval (uint32 a, uint32 b) public returns (uint) {
2       return a + b;
3   }
```

Fig. 1. An example of a signedness bug.

Integer bugs, like bugs related to callbacks, are a result of poorly-tested coding practices that occur in higher-level languages like Solidity, and not in the EVM opcodes. There are several integer bug checkers, such as Osiris [14] and Zeus [15], that can check for integer-based errors. Requiring these types of error checks before a contract can be listed, while not perfect, would identify and reduce the number of potential errors on the blockchain.

C. Bugs related to language and version inconsistencies

As indicated in the earlier discussion on arithmetic bugs, pre-0.4.0 versions of the Solidity software handle arithmetic errors differently than post-0.4.0 versions, which may result in unexpected behavior.

Although Solidity syntax resembles a mixture of C and JavaScript, it uses a variety of unique concepts specific to smart contract development and therefore may be unfamiliar for new developers [9]. There are several known issues with Solidity [16]. For example, EVM uses a memory model that is specific to the execution of smart contracts on a blockchain and differs from the traditional von Neumann architecture, which may cause confusion. EVM features four different types of memory, each with different properties and usage costs in terms of gas. While the stack and memory constructs are volatile and only hold values during execution of a contract, storage is persistent and part of Ethereum's world state, organized as a Patricia Merkle trie holding sets of persistent key/value pairs of all accounts. Storage is isolated from the other smart contracts and is the only way for a smart contract to save values across executions.

Solidity allows integers to be signed or unsigned, containing lengths of between 8 and 256 bits (in 8-bit increments) denoted, for example, as uint8 or int128. These resemble integer types in C and may lead developers to falsely assume that a uint8 occupies 8 bits in memory and an int128 occupies 128 bits. However, all integer types are represented in the EVM as a 256-bit big endian using twos-complement. Thus, integer typing in Solidity is inconsistent with that of the EVM, which can lead to programming errors. Explicit conversion between primitive types is possible, but the effects are poorly documented. Explicitly casting a signed negative integer to an unsigned one, for example, will not result in the absolute value as expected, but leave its bit-level representation intact.

Smart contracts are scripts stored on the blockchain and have a unique address. They are triggered by addressing a transaction to the smart contract script, which executes independently and automatically in a prescribed manner on every node in the network, making use of the data that was included in the triggering transaction. Coding bugs are bound to occur. For example confusion between == (equals) and != (not equals) allowed any user—apart from the smart contract creator—to freely enable and disable transactions of all ICX ERC-20 tokens on ICON's Ethereum smart contract (i.e., immobilizing an $800+ million blockchain) [17].

Although the Solidity language has become more robust in later (post 0.4.0) versions, poor non-centralized documentation indicates that these types of bugs will likely persist for some time. As it continues to gain prominence as the primary coding language on the largest smart contract platform, we anticipate that coders will exert greater influence to streamline the language.

D. Bugs related to execution inconsistencies

In Ethereum, gas is a unit that represents the relative amount of computational effort that it will take to execute certain operations on the blockchain. Every single operation that takes part in Ethereum, from a simple transaction or smart contract requires some amount of gas. Gas is used to calculate the fees paid to the network to execute an operation. When a smart contract is submitted, it has a pre-determined gas value (gas values for various EVM opcodes can be found in [2]). At the time of execution, each step of the contract requires a certain amount of gas, paying the network for the cost of bandwidth, cost of storage and cost of computation. Adding more than the required amount of gas can move the contract to the beginning of the queue and incentivize the transaction to be included in the next block. This incentive is analogous to a "higher priority" in other systems. Thus, in theory, it may be possible for collusion to occur and potentially starve other contracts from executing, though there have not been any examples of this occurring to date.

ERC-20 (Ethereum Request for Comment) is a technical standard used for smart contracts on the Ethereum blockchain for implementing tokens [18]. It defines a common list of rules for Ethereum tokens to follow within the larger Ethereum ecosystem, allowing developers to predict the interaction between tokens accurately. These rules include how the tokens are transferred between addresses and how data within each token can be accessed. Although at the time of this writing the clear majority of tokens issued on the Ethereum blockchain are ERC-20 compliant, interacting with tokens not compliant with this standard may introduce unforeseen errors. These non-compliant contracts can easily be checked before a transaction occurs, but as of the time of this writing, few if any smart contracts are known to make use of this information.

Last, as smart contracts become more ubiquitous, they attract more users with malicious intent [19]. This increased attraction is primarily due to the lack of regulation and the anonymity they provide. Smart contracts provide rich environments for Ponzi schemes. A Ponzi scheme is an unsustainable fraudulent investment operation in which the operator generates returns for older investors through revenue paid by new investors, rather than from legitimate business activities or profits of financial trading [20].

Detection of Ponzi schemes are difficult because of the anonymous, decentralized nature; these are contributing reason for their persistence in various forms for over a century. It is only possible to detect their presence by examining the smart contract code. Some researchers (e.g., [21) have recently developed tools that incorporate machine learning techniques to examine the flow of payments.

IV. Conclusions and Future Directions

Smart contracts, due to their self-executing, self-enforcing nature, provide great potential to disrupt numerous industries. However, as currently implemented, they are not without considerable risks and challenges. The results of these risks and challenges can be observed in the number of bugs, scams, and errors that have occurred since their inception in mid-2015.

As with any new technology, kinks are bound to occur. In this paper, we have discussed the three key properties that smart contracts need to contain (deterministic, isolated, and terminable), the four primary categories of bugs (callbacks, integer errors, language and version inconsistencies, and execution inconsistencies), and some methods that are being explored to maintain these principles and reduce the potential and severity of bugs. We focused on the development issues and did not discuss some identified issues with the Ethereum environment that are likely to limit its growth, such potential problems with scalability and the lack of a user-friendly development environment. We plan to expand on the topics covered here and address Ethereum and other emerging smart contract platforms in future work.

References

[1] V. Buterin. (2013). Ethereum: a next generation smart contract and decentralized application platform. [Online] Available: https://github.com/ethereum/wiki/wiki/White-Paper.

[2] V. Saini. (2018). ContractPedia: An Encyclopedia of 40 Smart Contract Platforms. [Online]. Available: https://hackernoon.com/contractpedia-an-encyclopedia-of-40-smart-contract-platforms-4867f66da1e5

[3] N. Szabo. (1994). Smart Contracts. [Online]. Available: http://szabo.best.vwh.net/smart.contracts.html

[4] N. Szabo. (1997). The Idea of Smart Contracts. [Online]. Available: http://szabo.best.vwh.net/smart_contracts_idea.html

[5] D. Puthal, N. Malik, S. P. Mohanty, E. Kougianos & G. Das, G. (2018). Everything You Wanted to Know About the Blockchain: Its Promise, Components, Processes, and Problems. IEEE Consumer Electronics Magazine, 7(4), 6-14.

[6] J. Little, O. Boukle-Hacene, and D. Guido. (2018) Ethereum VM (EVM) Opcodes and Instruction Reference. [Online] Available: https://github.com/trailofbits/evm-opcodes

[7] S. Tikhomirov. (2017). Ethereum: state of knowledge and research perspectives. [Online]. Available: https://hdl.handle.net/10993/32468.

[8] P. Daian. (2016). Analysis of the DAO exploit. [Online]. Available: http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/

[9] C.F. Torres & J. Schütte. (2018). Osiris: Hunting for Integer Bugs in Ethereum Smart Contracts. In Proceedings of the 34th Annual Computer Security Applications Conference (pp. 664-676). ACM.

[10] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, B., & B. Roscoe. (2018). ReGuard: finding reentrancy bugs in smart contracts. In Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings (pp. 65-68). ACM.

[11] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov. (2018). SmartCheck: Static Analysis of Ethereum Smart Contracts. In WETSEB'18: WETSEB'18:IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB 2018), Gothenburg, Sweden. ACM, New York. https://doi.org/10.1145/3194113.3194115

[12] Y. Hirai. (2016). Exception on overflow-Issue #796-ethereum/solidity. [Online] Available: https://github.com/ethereum/solidity/issues/796#issuecomment-253578925

[13] Secbit. (2018). A disastrous vulnerability found in smart contracts of BeautyChain (BEC). [Online]. Available: https://medium.com/secbit-media/a-disastrous-vulnerability-found-in-smart-contracts-of-beautychain-bec-dbf24ddbc30e

[14] O. Avan-Nomayo. (2018). Breaking News: Bug Discovered in ICON (ICX) Smart Contract – Token Transfers Disabled. [Online]. Available: https://ethereumworldnews.com/breaking-news-bug-discovered-in-icon-smart-contract-token-transfers-disabled/

[15] S. Kalra, S. Goel, M. Dhawan, M., & S. Sharma. (2018). Zeus: Analyzing safety of smart contracts. NDSS.

[16] P. Merriam. (2018). Solidity and Smart Contracts Gotchas. [Online] Available: https://populus.readthedocs.io/en/latest/gotchas.html

[17] N. Chong. (2018) A Single Incorrect Character Cripples ICON Blockchain, Bug Fix Implemented. [Online] Available: https://www.newsbtc.com/2018/06/17/single-incorrect-character-cripples-icon-blockchain-bug-fix-implemented/

[18] TheEthereum.wiki. (2017) ERC20 Token Standard [Online]. Available: https://theethereum.wiki/w/index.php/ERC20_Token_Standard

[19] Harris, Christopher G. (2018) "The risks and dangers of relying on blockchain technology in underdeveloped countries." In NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium, pp. 1-4. IEEE.

[20] Wikipedia. (2018). Ponzi scheme. [Online]. Available: https://en.wikipedia.org/wiki/Ponzi_scheme

[21] W. Chen, Z. Zheng, J. Cui, E. Ngai, P. Zheng, & Y. Zhou, Y. (2018). Detecting Ponzi Schemes on Ethereum: Towards Healthier Blockchain Technology. In Proceedings of the 2018 World Wide Web Conference on World Wide Web (pp. 1409-1418). International World Wide Web Conferences Steering Committee.