Smart Contracts Vulnerabilities: A Call for Blockchain Software Engineering?

Giuseppe Destefanis
School of Computer Science
University of Hertfordshire, UK
g.destefanis@herts.ac.uk

Michele Marchesi, Marco Ortu, Roberto Tonelli
University Of Cagliari, Italy
marchesi@unica.it
marco.ortu@diee.unica.it, roberto.tonelli@dsf.unica.it

Andrea Bracciali
University Of Stirling, UK
abb@cs.stir.ac.uk

Robert Hierons
Brunel University, UK
rob.hierons@brunel.ac.uk

***Abstract*—Smart Contracts have gained tremendous popularity in the past few years, to the point that billions of US Dollars are currently exchanged every day through such technology. However, since the release of the Frontier network of Ethereum in 2015, there have been many cases in which the execution of Smart Contracts managing Ether coins has led to problems or conflicts. Compared to traditional Software Engineering, a discipline of Smart Contract and Blockchain programming, with standardized best practices that can help solve the mentioned problems and conflicts, is not yet sufficiently developed. Furthermore, Smart Contracts rely on a non-standard software life-cycle, according to which, for instance, delivered applications can hardly be updated or bugs resolved by releasing a new version of the software.**

In this paper we advocate the need for a discipline of Blockchain Software Engineering, addressing the issues posed by smart contract programming and other applications running on blockchains. We analyse a case of study where a bug discovered in a Smart Contract library, and perhaps "unsafe" programming, allowed an attack on *Parity*, a wallet application, causing the freezing of about 500K Ethers (about 150M USD, in November 2017). In this study we analyze the source code of Parity and the library, and discuss how recognised best practices could mitigate, if adopted and adapted, such detrimental software misbehavior. We also reflect on the specificity of Smart Contract software development, which makes some of the existing approaches insufficient, and call for the definition of a specific Blockchain Software Engineering.

***Index Terms*—smart contracts; blockchain; software engineering;**

I. Introduction

Smart contracts are becoming more and more popular nowadays. They were first conceived in 1997 and the idea was originally described by computer scientist and cryptographer Nick Szabo as a kind of digital vending machine. He described how users could input data or value and receive a finite item from a machine (in this case a real-world snack or a soft drink).

More in general, *smart contracts* are self-enforcing agreements, i.e. contracts, as we intend them in the real world, but expressed as a computer program whose execution enforces the terms of the contract. This is a clear shift in the paradigm: untrusted parties demand the trust on their agreement to the *correct* execution of a computer program. A properly designed smart contract makes possible a crow-funding platform without the need for a trusted third party in charge of administering the system. It is worth remarking that such a third party makes the system centralized, where all the trust is demanded to a single party, entity, or organisation.

Blockchain technologies are instrumental for delivering the trust model envisaged by smart contracts.

In the example of a crowd-funding platform for supporting projects, the smart contract would hold all the received funds from a project's supporter (it is possible to pay a smart contract). If the project fully meets its funding goals, the smart contract will automatically transfer the money to the project. Otherwise, the smart contract will automatically refund the money to the supporters.

Since smart contracts are stored on a blockchain, they are immutable, public and decentralised. Immutability means that when a smart contract is created, it cannot be changed again and no one will be able to tamper with the code of a contract.

The decentralised model of immutable contracts implies that the execution and output of a contract is validated by each participant to the system and, therefore, no single party is in control of the money. No one could force the execution of the contract to release the funds, as this would be made invalid by the other participants to the system. Tampering with smart contracts becomes almost impossible.

The first Blockchain to go live was the Bitcoin's Blockchain [1] in 2009. It introduced the idea of programs used to validate agreement amongst untrusted parties: Bitcoin transactions are subject to the successful termination of a non-Turing complete program in charge of validating things like ownership and availability of the crypto money. The biggest blockchain that currently supports smart contracts is Ethereum, which was specifically created and designed with an extended execution model for smart contracts in 2014 [2]. Contracts in Ethereum can be programmed with Solidity, a programming language developed for Ethereum.

A few years down the line, several detrimental software misbehaviors, which caused considerable monetary loss and

community splits, have posed the problem of the correct design, validation and execution of smart contracts.

In this paper we advocate the need for a discipline of *Blockchain Software Engineering*, addressing the issues posed by smart contract programming and other applications running on blockchains. Blockchain Software Engineering will specifically need to address the novel features introduced by decentralised programming on blockchains. These will be discussed in more detail in the rest of this paper.

We consider a case study, the recent attack to the *Parity* wallet (2017). A bug discovered in a smart contract library used by the Parity application, caused the freezing of about 500K Ethers (see [3] for a summary).

We analyze the source code of Parity and the library, and reflect on the specificity of smart contract software development, noting some shortfalls of standard approaches to software development. We then discuss how recognized best practices in traditional Software Engineering could have mitigated, if adopted and adapted, such detrimental software misbehavior.

This paper aims to contribute a first step towards the definition of Blockchain Software Engineering.

II. Background

In this section we briefly introduce the blockchain and smart contracts technology, their execution environment and model. Since our study is focused on the Ethereum platform we will use it as example but the concepts presented here are of general validity.

A. Decentralized Ledgers

A blockchain is essentially a shared ledger that stores transactions, holding pieces of information, in a decentralized peer-to-peer network. Nodes are called *miners* and each one maintains a consistent copy of the ledger. Transactions are grouped together into blocks, each hash-chained with the previous block. Such a data structure is the so called *blockchain*, shown in Figure 1.

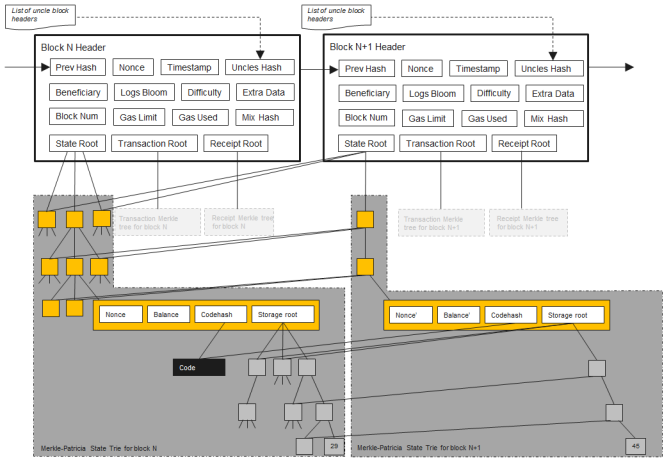

Fig. 1. Blockchain and Ethereum architecture. Each block of the chain consists of a large number of single transactions.

Miners use a consensus protocol in order to agree on the validity of each block, called *Nakamoto Consensus Protocol* [1]. At any time miners group their choice of incoming new transactions in a new block, which they intend to add to the public blockchain. Nakamoto consensus uses a probabilistic algorithm for electing the miner who will publish the next valid block in the blockchain. Such a miner is the one who solves a computationally demanding cryptographic puzzle. Such a procedure is called *proof-of-work*. All other miners verify that the new block is correctly constructed (e.g. no virtual coin is spent twice) and update their local copy of the blockchain with the new block. Bitcoin transactions essentially record the transfer of coins from one address, a wallet say, to another one. Differently, Ethereum transactions also include *contract-creation* transactions and *contract-invoking* transactions. The former ones record a smart contract on the blockchain, and the latter ones cause the execution of a contract functionality (which enforces some terms of the contract). We refer the reader to the original white papers of Bitcoin and Ethereum [1], [2] for further details.

B. Ethereum Smart Contracts

A *Smart Contract* (SC) is a *full-fledged program* stored in a blockchain by a *contract-creation* transaction. A SC is identified by a *contract address* generated upon a successful creation transaction. A blockchain state is therefore a mapping from addresses to accounts. Each SC account holds an *amount of virtual coins* (Ether in our case), and has its own private *state* and *storage*. An Ethereum SC account hence typically holds its executable code and a state consisting of:

- private storage
- the amount of virtual coins (Ether) it holds, i.e. the contract *balance*.

Users can transfer Ether coins using transactions, like in Bitcoin, and additionally can *invoke* contracts using *contract-invoking* transactions. Conceptually, Ethereum can be viewed as a huge *transaction-based state machine*, where its state is updated after every transaction and stored in the blockchain.

A Smart Contract's source code manipulates variables in the same way as traditional imperative programs. At the lowest level the code of an Ethereum SC is a stack-based bytecode language run by an Ethereum virtual machine (EVM) in each node. SC developers define contracts using high-level programming languages. One such language for Ethereum is Solidity [4] (a JavaScript-like language), which is compiled into EVM bytecode. Once a SC is created at an address X, it is possible to invoke it by sending a contract-invoking transaction to the address X. A contract-invoking transaction typically includes:

- payment (to the contract) for the execution (in Ether).
- input data for the invocation.

1) Working Example: Figure 2 shows a simple example of SC reported in [5], which rewards anyone who solves a problem and submits the solution to the SC.

A *contract-creation* transaction containing the EVM bytecode for the contract in Figure 2 is sent to miners. Eventually,

```
contract Puzzle {                                                    1
                                                                     2
  address public owner ;                                             3
  bool public locked ;                                               4
  uint public reward ;                                               5
  bytes32 public diff ;                                              6
  bytes public solution ;                                            7
                                                                     8
  function Puzzle ()  {// constructor                                9
    owner = msg.sender ;                                             10
    reward = msg.value ;                                             11
    locked = false ;                                                 12
    diff = bytes32 (11111); // pre-defined                           13
        difficulty
  }                                                                  14
                                                                     15
  function (){ // main code , runs at every                          16
      invocation
    if ( msg.sender == owner ) { // update reward                    17
      if ( locked )                                                  18
        throw ;                                                      19
      owner.send(reward);                                            20
      reward = msg.value ;                                           21
    } else if ( msg.data.length > 0) {                               22
      // submit a solution                                           23
      if ( locked ) throw ;                                          24
      if ( sha256 ( msg.data ) < diff ) {                            25
        msg.sender.send(reward); // send reward                      26
        solution = msg.data ;                                        27
        locked = true ;                                              28
      }                                                              29
    }                                                                30
  }                                                                  31
}                                                                    32
```

Fig. 2. Smart Contracts example.

the transaction will be accepted in a block, and all miners will update their local copy of the blockchain: first a *unique* address for the contract is generated in the block, then each miner executes locally the *constructor* of the **Puzzle** contract, and a local storage is allocated in the blockchain. Finally the EVM bytecode of the anonymous function of **Puzzle** (Lines 16+) is added to the storage.

When a *contract-invoking* transaction is sent to the address of **Puzzle**, the function defined at Line 16 is executed by default. All information about the sender, the amount of Ether sent to the contract, and the input data of the invoking transaction are stored in a default input variable called *msg*. In this example, the *owner* (namely the user that created the contract) can update the *reward* (Line 21) by sending Ether coins stored in `msg.value` (`if` statement at Line 17), after sending back the current *reward* to the *owner* (Line 20).

In the same way, any other user can submit a solution to **Puzzle** by a *contract-invoking* transaction with a *payload* (i.e., `msg.data`) to claim the reward (Lines 22-29). When a correct solution is submitted, the contract sends the reward to the sender (Line 26).

2) Gas System: It is worth remarking that a SC is run on the blockchain by each miner deterministically replicating the execution of the SC bytecode on the local copy of the blockchain. This, for instance, implies that in order to guarantee coherence across the copies of the blockchain, code must be executed in a strictly deterministic way (and therefore, for instance, the generation of random numbers may be problematic).

Solidity, and in general high-level SC languages, are Turing complete in Ethereum. In a decentralised blockchain architecture Turing completeness may be problematic, e.g. the replicated execution of infinite loops may potentially *freeze* the whole network.

To ensure fair compensation for expended computation efforts and limit the use of resources, Ethereum pays miners some fees, proportionally to the required computation. Specifically, each instruction in the Ethereum bytecode requires a pre-specified amount of *gas* (paid in Ether coins). When users send a *contract-invoking* transaction, they must specify the amount of gas provided for the execution, called *gasLimit*, as well as the price for each gas unit called *gasPrice*. A miner who includes the transaction in his proposed block receives the transaction fee corresponding to the amount of gas that the execution has actually burned, multiplied by *gasPrice*. If some execution requires more gas than *gasLimit*, the execution terminates with an exception, and the state is rolled back to the initial state of the execution. In this case the user pays all the *gasLimit* to the miner as a counter-measure against resource-exhausting attacks [6].

III. Case Study and Methodology

Recently Ethereum suffered a supposedly involuntary hack, in which an inexperienced developer froze multiple accounts managed by the Parity Wallet application. The hack has suddenly risen to the news since the amount of Ether coins in the frozen account was estimated to be 513,774.16 Ether (equivalent to 162M USD at the time). In November 2017 the hack became (in)famously known as the *Parity Wallet hack*. This was the result of a single library code deletion.

This case replicated a similar problem exploited by another hacker on the same Parity Wallet code, a few months earlier (July 2017). In that case, a multi-signature wallet was hacked and set in control of a single owner, who acquired all the Ether coins of that single wallet.

The Parity Wallet hack represents a paradigmatic example of problems that may currently occur in the development of smart contracts and blockchain-related software in general. Such problems are associated to the lack of suitably standardised best practices for blockchain software engineering.

In the rest of this paper, we will analyse the Parity Wallet hack case by applying static code analysis to the Parity library. We aim to understand the code structure and, more importantly, the link between smart contracts and the functions defined in smart contract libraries. After analyzing the Solidity code of the Wallet, we will outline the events that ended up with more than 500k Ether frozen. Then, we will elaborate on viable solutions from the perspective of Software Engineering. These represent potential general solutions for cases analogous to the Parity Wallet hack.

IV. Structure and Functionality of Parity

Parity is an Ethereum client that is integrated directly into web browsers. It allows the user to access the basic

Ether and token wallet functions. It is also an Ethereum GUI browser that provides access to all the features of the Ethereum network, including DApps (decentralised applications). Parity also operates as an Ethereum full node, which means that the user can store and manage the blockchain on his own computer. It is a complex and critical decentralised application.

Solidity and the EVM provide three ways to call a function on a smart contract: CALL, CALL-CODE, and DELEGATE-CALL. The former is a call to a function that will be executed in the environment of the called contract. The other two calls execute the called code in the caller environment. Many library calls on Ethereum are implemented with DELEGATE-CALL, typically by deploying a contract that serves as a library: the contract has functions that anyone can call, and these may be used, for instance, to make changes in the storage of the calling contracts. Solidity has some syntactic constructs which allow *libraries* offering DELEGATE-CALLs to be defined and "imported" by other contracts. However, at the EVM level the library construction disappears, and DELEGATE-CALLs and other calls are actually deployed as smart contract functionalities.

1) Statically linked libraries: It is possible to embed all the library code in a smart contract, e.g. the multi-signature wallet contract itself, instead of using DELEGATE-CALLs to an external contract. In a sense, this would be similar to the standard static linking of libraries. However, statically linked code increases the gas cost of contract deployment (space also has a cost).

2) Parity library contract: Parity made the choice to adopt library-driven smart contract development for their multi-signature wallets. That is, Parity initially deployed a multi-signature contract as their library, and all the other Parity multi-signature wallets referenced that single library contract for all their functionality. The library itself was actually a properly working multi-signature wallet. In hindsight, it probably shouldn't have been.

All the Parity multi-signature wallets (except for the library one) reference the library by declaring the following constant[1]:

```
address constant _walletLibrary =                    1
0x863df6bfa4469f3ead0be8f9f2aae51c91a907b4            2
```

Since it is a constant, it is generated at compile time, meaning it's permanently stored in "code", not in "storage". The value would be the address of the library to DELEGATE-CALL on. By running

```
eth.getCode(walletAddress)                           1
```

on one of the affected wallets (`walletAddress`), it is still possible to see the address of the now-dead library at the line code of index 422.

Another observation is that it would probably be better practice to allow the owners of the wallets to change the linked library, instead of coding it in the bytecode.

[1]https://medium.com/crypt-bytes-tech/parity-wallet-security-alert-vulnerability-in-the-parity-wallet-service-contract-1506486c4160

V. Analysis of the Attack

In this section we report a summary of the description of the attack presented on "ethereum.stackexchange.com" at the link https://ethereum.stackexchange.com/questions/30128/explanation-of-parity-library-suicide.

Remarkably, the attack that we are discussing was announced by a post of the supposedly unaware author: *"I accidentally killed it."*[2] The author took control of a library contract, killed it, obliterating functionality for ∼500 multi-signature wallets and effectively, irreversibly freezing ∼$150M. A hard fork would be required to restore the contract and/or return funds.

It is important to highlight that the library we are considering was a working wallet. However, it had *not been initialized* since it was a *library* contract and the variable `only_uninitialized` had not been set. The attack could have been avoided if, after Parity deployed the library contract, it would have called `initWallet-once()` to claim the contract and set the uninitialized variables, including `owner`.

Anyone could then call the function *initWallet* on the library contract. As the "hacker" did. Such a call, amongst other things, sets the caller, i.e. the hacker, as the owner of the contract being initialised. It is worth remarking that such a call is perfectly legal and it just initializes a wallet which has not been yet initialized. At this point, the owner of the library contract (e.g., an hacker), can call any privileged function, amongst which `kill()`. The kill function calls `suicide()`, which is now being replaced by `self-destruct`. The suicide function sends the remaining funds to the owner, destroying the contract and clearing its storage and code. Figure 3 shows the diagram of the functions and their dependencies for the Parity smart contract library defined at the address `0x863df6bfa4469f3ead0be8f9f2aae51c91a907b4`

Every call to the library will now return false and the multi-signature wallet contracts relying on the library contract code would get zero (with DELEGATE-CALL). The contracts still hold funds, but all the library code is set to zero. The multi-signature wallets are locked and the majority of the functionalities depend on the library which returns zero for every function call.

Indeed, after having killed the library contract, any other contracts depending on the killed library queried with

```
isowner(any_addr)                                    1
```

return `TRUE`, as a consequence of the delegate call made to a dead contract (the hacker tried this, to allegedly test the exploit).

The Ethereum Transaction that tracks the kill call is

```
0x47f7cff7a5e671884629c93b368cb18f58a                1
993f4b19c2a53a8662e3f1482f690                        2
```

Wallets deployed before July 19 used a different library contract with a similar initWallet bug, but the library contract

[2]https://github.com/paritytech/parity/issues/6995#issuecomment-342409816

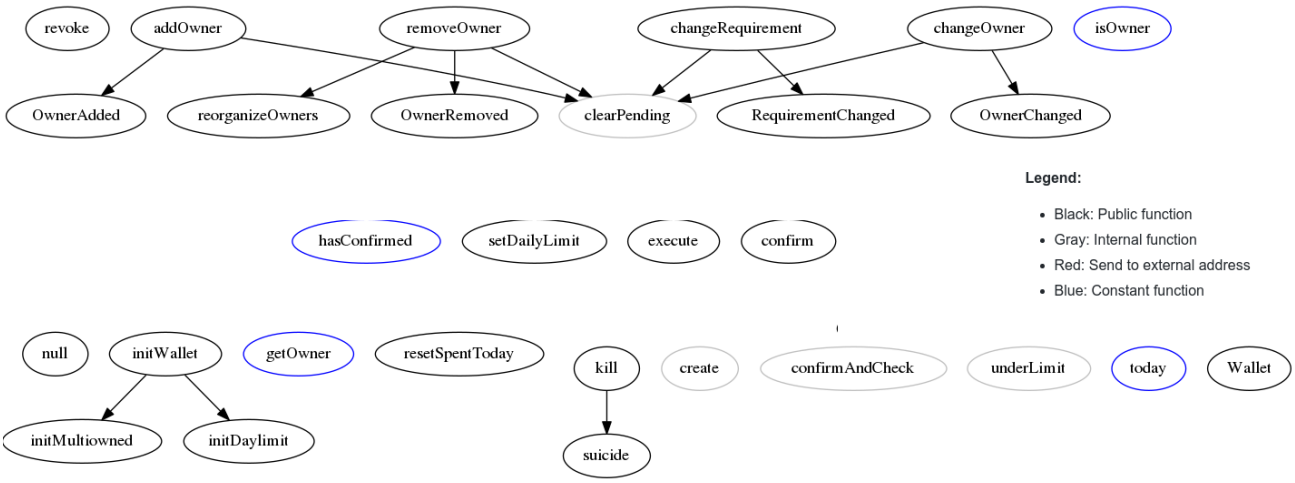

Fig. 3. Parity Wallet Dependency Graph

could not be taken over in a similar fashion as it had already been initialized by the developers at Parity.

The choice of defining the Wallet library as a contract instead of as a library, with the actual wallets making simple DELEGATE-CALLs to this linked smart contract, also needs to be confronted with the recommended practice of clearly defining libraries as such. Such a choice, makes the library contract behave more like a Singleton than a proper Library.

The only way to restore wallets' functionalities would be a *hard fork* in order to re-enable the library code. It is worth recalling that a hard fork would require an agreement by the majority of the miner community and their coordinated effort to develop an alternative branch of the blockchain, where the hack has basically never occurred. Although there have been cases in which this has been done, e.g. for the DAO attack [7], such a recovery strategy is an extraordinary event of extremely difficult realisation, which bears disastrous consequences - e.g. the cancellation of "happened" independent transactions - and that cannot currently be considered a routine error-correction practice.

VI. Best Practices That Could Have Helped

Smart Contracts security is an open research field [5]. The development of bug-free source code is still an utopia for traditional software development, after decades of analysis and development of engineering approaches. Error freedom is even more daunting for blockchain software development, which started less then a decade ago. In this section we discuss approaches that could have helped in mitigating the effects of the attack, drawing from accepted best practices in traditional software engineering. It is worth remarking here that vulnerabilities like the one leading to the Parity attack had been highlighted in literature, e.g. [8], a fact that strengthens even more our call for the adoption of standard and best practices in Blockchain Software Engineering.

A. Anti-patterns

An anti-pattern is a common response to a recurring problem that is usually ineffective and risks being highly counterproductive.[3] We have identified three anti-patterns in our case study that are responsible for the issue under analysis.

- The creation of a SM, that serves as full-fledged library, which is then left uninitialized.
- The creation of SMs that depend on external SM used as a library, and the address of such external library is hard-coded in the SMs and cannot be updated.
- The used SM library includes the definition of a public function that might call destroying functionality, such as *suicide*.

On the contrary, a pattern to be used is

- To allow SCs to re-address other SCs code whenever these are used as a library.

Such a strategy would also enable SCs to reference new libraries that have been deployed in more recent blocks. Furthermore, the strategy can be exploited for debugging purposes, refactoring, introduction of new features, and, in general, for purposes similar to versioning in traditional software engineering.

For example, faults could be so corrected in SC code and the corrected version of the same SC can be successively re-deployed and accepted by miners. Once provided with the address of the debugged contract, the very same contracts that were calling the faulty version can call the debugged contract. A similar scenario can be used for any issue resolution as in traditional versioning systems.

Such a solution applies to the present case study, in the hypothesis that the Wallet library address would have been saved into the private storage of each Parity wallet, and

[3]Definition from https://en.wikipedia.org/wiki/Anti-pattern. The term *anti-pattern* was coined in 1995 by Andrew Koenig.

managed through setter and getter methods, instead of being hard coded.

B. Testing

Testing smart contract is challenging and critical, because once deployed on the blockchain they become immutable, not allowing for further testing or upgrading. At present, to the best of our knowledge, there is not a testing framework for Solidity, like e.g. JUnit for the Java language, meaning that every smart contract has to be tested manually.

The nature of smart contracts introduces at least two complications to testing: an application may be critical and it is very difficult to update an application once deployed. As a result, it is desirable to use robust testing techniques. Manual test generation is likely to form an important component but inevitably is limited; there is a need for effective automated test generation (and execution) techniques.

Currently available options to test contracts are:

- Deploy the contract to live (the real) Ethereum main network and execute it. This costs about 5 minutes and real money to deploy and execute (slow, public(dangerous), $$$).
- Deploy the contract to the test-net Ethereum network (for developers usage) and execute it. This costs about 2 minutes to deploy and free ether (slow, public(dangerous), free(no real money))
- Deploy the contract to an Ethereum network (local) simulator and execute it. This costs about 3 seconds and is free (fast, private(nice!!), free), but limited interaction and no realistic test of network related issues.

There are many automated test generation approaches that might have value. A number of these use formal approaches, such as those based on symbolic execution (e.g. [9]), or search-based methods (e.g. [10]) to produce test cases that provide code coverage. It is unclear whether such techniques would have found the attack on Parity since the attack involved a sequence of operations and code coverage techniques typically aim to cover smaller structures, such as branches in the code[4]. Code coverage approaches may still have value but it appears that they are not sufficient on their own.

An alternative, and complementary, approach is to base test automation on a model; an approach that is typically called model-based testing (MBT) (see [11] for a survey). Some MBT techniques aim to cover the model but there are also more rigorous approaches that generate test cases that are guaranteed to find certain classes of faults (defined by a fault model) or that are guaranteed to find all faults if certain well-defined conditions (test hypotheses) hold [12]. Such approaches provide a trade-off: as one weakens the assumptions or widens the class of faults, the cost of testing increases. There is also the scope to utilize formal verification techniques in order to reason about the underlying assumptions made by these techniques or, indeed, to find faults.

[4]It is possible to require, for example, the coverage of paths but such approaches tend not to scale.

We have seen that smart contracts are state-based and so it would be natural to use state-based models in MBT, allowing the use of a wide range of test automation techniques. There appears to be potential for MBT approaches to find faults such as the one that resulted in the Parity attack. First, the process of producing a model might have led the developers to consider what happens if a user calls a library function without it being initialized. Second, the attack involved a particular (short) sequence of operations and state-based MBT techniques focus on the generation of such sequences. Naturally, the effectiveness of MBT depends on the model and also the fault model (or test hypotheses) used. An interesting challenge is to explore smart contracts and their faults in order to derive appropriate fault models or test hypotheses.

VII. Road-map to BOSE

The Parity wallet case study clearly showed that a Blockchain-Oriented Software Engineering (BOSE) [13], [14], [15], [16], [17] is needed to define new directions to allow effective software development. New professional roles, enhanced security and reliability, modeling and verification frameworks, and specialized metrics are needed in order to drive blockchain applications to the next reliable level. At least three main areas to start addressing have been highlighted by our analysis of a specific case of study:

- Best practices and development methodology
- Design patterns
- Testing

The aim of BOSE is to create a bridge between traditional software engineering and blockchain software development, defining new ad-hoc methodologies, fault analysis [18], patterns [19], [20], quality metrics, security strategies and testing approaches [21] capable of supporting a novel and disciplined area of software engineering.

VIII. Conclusions

In this paper, we presented a study case regarding the Parity smart contract library. The problem resulted from poor programming practices that led to the situation in which an anonymous user was able to accidentally (it is not clear if he did it on purpose) freeze about 500K Ether (150M USD on November 2017).

We investigated the case, analyzing the chronology of the events and the source code of the smart contract library. We found that the vulnerability of the library was mainly due to a negligent programming activity rather than a problem in the Solidity language.

The vulnerability was exploited by the anonymous user in two steps. First the attacker was able to become the owner of the smart contract library (because it was created and left uninitialized), then the attacker did nothing more than calling the initialization function. After that the suicide function was called, which killed the library, leading to the situation in which it was not possible to execute functionality on the smart contracts created with the library, because all the delegate calls ended up in the dead smart contract library. This case

clearly demonstrated a need for Blockchain Oriented Software Engineering in order to prevent, or mitigate such scenarios.

The aim for BOSE is to pave the way for a disciplined, testable and verifiable smart contract software development.

REFERENCES

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[2] "Ethereum foundation. the solidity contract-oriented language." https://github.com/ethereum/solidity., 2014.

[3] "A postmortem on the parity multi-sig library self-destruct," https://paritytech.io/a-postmortem-on-the-parity-multi-sig-library-self-destruct/, 2017.

[4] "Ethereum foundation. ethereum original white paper." https://github.com/ethereum/wiki/wiki/White-Paper, 2014.

[5] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 254–269.

[6] L. Luu, J. Teutsch, R. Kulkarni, and P. Saxena, "Demystifying incentives in the consensus computer," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 706–719.

[7] D. Siegel, "Understanding the dao attack," http://www.coindesk.com/understanding-dao-hack-journalists, 2016.

[8] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts sok," in *Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204*. New York, NY, USA: Springer-Verlag New York, Inc., 2017, pp. 164–186. [Online]. Available: https://doi.org/10.1007/978-3-662-54455-6_8

[9] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Commun. ACM*, vol. 56, no. 2, pp. 82–90, 2013. [Online]. Available: http://doi.acm.org/10.1145/2408776.2408795

[10] M. Harman and P. McMinn, "A theoretical and empirical study of search-based testing: Local, global, and hybrid search," *IEEE Trans. Software Eng.*, vol. 36, no. 2, pp. 226–247, 2010. [Online]. Available: https://doi.org/10.1109/TSE.2009.71

[11] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. J. Krause, G. Lüttgen, A. J. H. Simons, S. A. Vilkomir, M. R. Woodward, and H. Zedan, "Using formal specifications to support testing," *ACM Comput. Surv.*, vol. 41, no. 2, pp. 9:1–9:76, 2009. [Online]. Available: http://doi.acm.org/10.1145/1459352.1459354

[12] M. Gaudel, "Testing can be formal, too," in *TAPSOFT'95: Theory and Practice of Software Development, 6th International Joint Conference CAAP/FASE, Aarhus, Denmark, May 22-26, 1995, Proceedings*, ser. Lecture Notes in Computer Science, vol. 915. Springer, 1995, pp. 82–96.

[13] A. Pinna, R. Tonelli, M. Orrú, and M. Marchesi, "A petri nets model for blockchain analysis," *arXiv preprint arXiv:1709.07790*, 2017.

[14] S. Porru, A. Pinna, M. Marchesi, and R. Tonelli, "Blockchain-oriented software engineering: challenges and new directions," in *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press, 2017, pp. 169–171.

[15] H. Rocha, S. Ducasse, M. Denker, and J. Lecerf, "Solidity parsing using smacc: Challenges and irregularities," in *Proceedings of the 12th Edition of the International Workshop on Smalltalk Technologies*, ser. IWST '17. New York, NY, USA: ACM, 2017, pp. 2:1–2:9. [Online]. Available: http://rmod.inria.fr/archives/workshops/Roch17a-IWST-SolidityParser.pdf

[16] S. Bragagnolo, H. Rocha, M. Denker, and S. Ducasse, "Smartinspect: Smart contract inspection technical report," Inria Lille-Nord Europe, Technical Report, Dec. 2017. [Online]. Available: http://rmod.inria.fr/archives/reports/Roch17b-TR-SmartInspect.pdf

[17] R. Tonelli, G. Destefanis, M. Marchesi, and M. Ortu, "Smart contracts software metrics: a first study," *arXiv preprint arXiv:1802.01517*, 2018.

[18] M. Ortu, G. Destefanis, S. Swift, and M. Marchesi, "Measuring high and low priority defects on traditional and mobile open source software," in *Proceedings of the 7th International Workshop on Emerging Trends in Software Metrics*. ACM, 2016, pp. 1–7.

[19] G. Destefanis, R. Tonelli, G. Concas, and M. Marchesi, "An analysis of anti-micro-patterns effects on fault-proneness in large java systems," in *Proceedings of the 27th Annual ACM Symposium on Applied Computing*. ACM, 2012, pp. 1251–1253.

[20] G. Destefanis, R. Tonelli, E. Tempero, G. Concas, and M. Marchesi, "Micro pattern fault-proneness," in *Software engineering and advanced applications (SEAA), 2012 38th EUROMICRO conference on*. IEEE, 2012, pp. 302–306.

[21] S. Counsell, G. Destefanis, X. Liu, S. Eldh, A. Ermedahl, and K. Andersson, "Comparing test and production code quality in a large commercial multicore system," in *Software Engineering and Advanced Applications (SEAA), 2016 42th Euromicro Conference on*. IEEE, 2016, pp. 86–91.