Analysis on attacks and defenses in Ethereum Ecosystem

TSO CHUN FAI 20012065g*

Dec 2, 2020

Contents

1 Introduction **3**
1.1 Ethereum 3

2 Problem Statement **4**
2.1 Problem definition and assumption 4
2.2 Challenging issues in solving the problem 4
2.2.1 Non-standard semantic behaviours 4
2.2.2 EVM bytecode is hard to analyze 4
2.2.3 Smart contract security properties are difficult to reason . 4

3 Comparison of the solutions proposed in the 5 papers **5**
3.1 Introduce the solutions proposed 5
3.1.1 Topic paper 5
3.1.2 Semantic Framework 5
3.1.3 Securify 6
3.1.4 Manticore 7
3.1.5 ContractFuzzer 8
3.2 Summarize the novelty of each solution 9
3.2.1 Topic paper 9
3.2.2 Semantic Framewrok 9
3.2.3 Securify 9
3.2.4 Manticore 9
3.2.5 ContractFuzzer 10
3.3 Adavantages and disadvantages of the solution proposed 10
3.3.1 Topic paper 10
3.3.2 Semantic Framework 10

*CERTIFICATE OF ORIGINALITY: I hereby declare that this report is my own work and that, to the best of my knowledge and belief, it reproduces no material previously published or written, nor material that has been accepted for the award of any other degree or diploma, except where due acknowledgement has been made in the text.

3.3.3 Securify 11
3.3.4 Manticore 12
3.3.5 ContraFuzzer 12
3.4 Methodology and results 12
3.4.1 Topic Paper 12
3.4.2 Semantic Framework 13
3.4.3 Securify 13
3.4.4 Manticore 13
3.4.5 ContraFuzzer 14

4 Summary **15**
4.1 Limitations 15

5 Conclusion **15**

1 Introduction

1.1 Ethereum

Ethereum is a cryptographic currency and like the more well-know BitCoin, it is built upon a blockchain, which is like a public ledger where all the transactions on the ledger are verified and publicly available. Network participants can push transactions to the blockchain network. These transaction will then be grouped and appended to the blockchain using some consensus mechanism. The Ethereum used the proof of state system right not. Transactions can be carried out by human or smart contract which will then be executed in the Ethereum Virtual Machine (EVM). EVM is quasi Turing complete as the execution could be restricted by the gas limit which essentially limits the number of execution steps. These smart contracts are usually written in a prgramming language called Solidity. Similar to object- oriented programming, Solidity is so-called "contract-oriented" [3]. The syntax of Solidity is similar to JavaScript, with some primitives accounting for the distributed nature of Ether. Examples of these primitives are `msg.sender` etc.[3]

With the upgrade of the Ethereum to Ethereum 2.0, the platform promised to be more scalable and secure, again gaining the world's attention. In recent year, several huge incidents made us question whether the platform is ready for wider adoption. To answer this we will need to study the platform, especially the `smart contract` more thoroughly. The real question, however, seemed to be *how can we study the security of ethereum smart contract?*

Smart contracts are just like computer programmes, having similar kinds of vulnerability such as integer overflow, reentrancy and call injection etc [12]. Afterall, most smart contracts are written in high-level programming language, Solidity being the most popular one. These codes will then be compile to low-level bytecode which would then be deployed to the blockchain, invoked by users or smart contracts and be executed by the Ethereum Virtual Machine (EVM) As such, different kinds of technique used to analysis codes could also be used to study smart contracts.

The stake is high here because in the infamous DAO vulnerability has led to a financial losses or more than sixty million USD and if we really want to see more real-world application of the smart contracts, we must safeguard or at least have the tool to enable us to safeguard the plarform from hacker alike.

In this report, we will be looking at several technique for detecting vulnerability: The mother of all defenses strategy, without which we will be fighting in the dark.

2 Problem Statement

2.1 Problem definition and assumption

An effective security analyzer for smart contract is needed. It has to be scalable, effective and versatile in order to cope with the ever-changing landscape of the attack-defense game.

2.2 Challenging issues in solving the problem

2.2.1 Non-standard semantic behaviours

In[3], the authors noted that the semantics in the Ethereum yellow paper has exhibited several under-specifications and does not follow the standard approach for the specification of program semantics, and thus hinders the program verification.

2.2.2 EVM bytecode is hard to analyze

[3] EVM bytecode is a stack-based code featuring dynamic code creation and invocation with little static information. All of these features make it very difficult to analyze with.

2.2.3 Smart contract security properties are difficult to reason

The fundamental issue here is that our understanding what properties should a smart contract has in order to be safe is immature. Smart contract is basically a programe, then we can ask ourselves what properties should a computer programe in general should possess in order to be safe? The question can be answered in many different levels of abstraction from a single assignment statement to the whole inter-module function calls (in smart contract context, this might assemble inter-contract transactions). So how can/should we be answering these questions? Or better still, can we propose a formalized framework for security properties in smart contract.

Given all these difficulties, it would be naive of us to think solutions could be easily found, therefore we would be explore detection mechanism from different "paradigms": Static code analysis, Dynamc code analysis, Signature matching analysis, Semantic Framework analysis and Fuzzer analysis. It is with all these different angles, we can truly see and learn more about the problem we are facing.

3 Comparison of the solutions proposed in the 5 papers

3.1 Introduce the solutions proposed

3.1.1 Topic paper

[12]The paper proposed a pattern matching framework for analyzing real-world adoptions of attacks and defenses. It basically matches execution traces outputted by "uninstrumented EVM" with known adversarial transaction signatures. This matching of signature involves two stages: Adversarial action and adversarial consequence.

Adversarial action clause The paper proposed an action clause of the adversarial signature and match it against the interactions in the contract and decide whether a tranaction is adversarial. For each transaction or contract, an action tree is constructed to represent all the interact-contract interactions (function calls, contract destruction etc). We will then locate the adversarial transactions by matching action clause to action trees

Adversarial result clause Similar to the action clause, it models the ownership transfers between contracts. A result graph, similar to the action tree, is constructed to represent the transfers between contracts for each of the transaction or contract. Finally we will do the matching like the above.

Transaction-centric vs Contract-centric In constructing the action tree and result graph for matching signatures, the author made a distinction between Transaction-centric and Contract-centric construction in which different adversarial transactions should be detected using different construction. For example, Call injection, reentracy and Integer Overflow will be classified as Transaction-centric.

Signature Generation The process involve two parts: Invariant extration and human reasoning. The first step is to extract common node and edges from action trees and result graph of known adversarial transactions. The second is then to reason them manually.

3.1.2 Semantic Framework

In[3], a complete small-step semantics of EVM bytecode formalized in the F* proof assistant[8]. Executable code produced is then successfully validated against the Ethereum official tests.

Small-Step Relation The paper proposed to a small-step relation that specify how a call stack change from one state to the next one-step-ahead state in the transaction environment. Starting from these two sets, the authors further proposed a full grammer for these two sets and six other sets in order to formalize the framework of the Small-Step Relation. For details, please refer to the Figure1 in [3]

Small-Step Rules The paper presented a selection of these rules to illustrate the features of the semantics. These selected rules includes `ADD`, `CALL` etc. Please refer to the paper for details, due to the space limitation here.

Security properties The authors proposed four security property which could be exhaustively categorized all classes of bugs known by other published works. These properites are

- Call integrity Bug: Reentrancy, Call to the unknow
- Atomicity Bug: Mishandled exceptions
- Independence of mutable account state Bug: Transaction order dependency, Unpredictable state
- Independence of transaction environment Bug: Timestamp dependancy, Time contraints, Generating randomness

3.1.3 Securify

[7]The paper proposed a framework called Securify. It involves decompiling the contract bytecodes into semantic facts using the smart contract's dependency graph. A dependency graph is a directed graph for representaiton of dependencies within objects. It would then encode the graph in Datalog, which is a logic programming language. Some scalable Datalog solvers which could analyze the code within seconds can then be used to analyze the code

How to analyze: A simplified example Securify would provide two kinds of of pattern for the solver to analyze against: Compliance patterns and violations patterns. Given as an motivating example, the paper illustrated the approach with a simplified version of the Parity wallet attack: Any users could call a `initWallet` function and store any arbitrary address in the contract field `owner`. Second, the attacker will then call the `withdraw` to steal the ether. Here a security issue is identified, that is the `owner` field should not be writable by other user.

Example To translate this property to a pattern expressed in dependency graph, we can simple enforce the dependency that the execution of the field assignment `owner = _owner` must depend on the value return by the `caller` instruction, which should return the address of the transaction sender.

Inference of semantic facts First, the system will extract a set of base facts from EVM bytecode into static-single assignment form of stackless representation. It is noted that many opcodes in the EVM assembly instructions will be eliminated (for example Block information: number, timestamp, gaslimit). This set of base facts will then be given to Datalog to produce a set of semantic facts that are inferred with the base facts given to the Datalog program where a set of inference rules are set by the authors. The output of the program is semantic facts. There are two kinds of these semantic facts: Flow-Dependency Predicates and Data-Dependency predicates.

Construction of patterns Securify will then construct patterns according to syntax defined by the athors. The pattern can include instructions, the predicates defined above and some custom composite patterns of the instructions and predicates. A set of compliance and violation security patterns will be written in this DSL and be compared against the semantic facts output of the contract.

Security patterns Sever property, each with compliance and violation, are used to construct the set of security patterns. Some of these property is established using existing vulnerability

3.1.4 Manticore

Manticore[6] is a dynamic symbolic execution framewrok for analyzing not only smart contract but also binaries. Symbolic execution is a method of conduct analysis of a program to find out what inputs are required for each parts of a program to execute[11]."Dynamic symbolic execution for automated test generation consists of instrumenting and running a program while collecting path constraint on inputs from predicates encountered in branch instructions,and of deriving new inputs from a previous path constraint by an SMT (Satisfiability Modulo Theories) solver in order to steer next executions toward new program paths."[2] These path constraints or predicates (as called by the paper) can be used to identify all the program states that can be triggered during runtime and these states are guaranteed to be reproducible, therefore producing no false positives.

Architecture Manticore contains, for our purpose here, Core Engine, Ethereum Execution Module , SMT, Event System, and API. We will be focusing on the Core Engine and the Ethereum Execution Module.

Core Engine The Core Engine manages program states with a simple state life cycle which have three states: `Ready`, `Busy` and `Terminated`. It has two events: `Termination` and `Concretization`. States are objects representing the state of a program at a given point in the execution. The engine will select and execute a `Ready` state, turning it to `Busy` which would then go back to

either `Ready` or to one of the above events. `Termination` event will change the state to `Terminate` while `Concertization` event will change a symbolic object to one or more concrete values, like assigning a numercial value to a variable.

Ethereum Execution Module [6] noted smart contracts are basically state machines with some salient features such as "gas" cost for executing instructions. These differences do not preclude the Core Engine from suppor for Ethereum.

Ethereum Symbolic Execution Author noted that, in Ethereum, smart contracts receives input as transactions which consist of a value and a data buffer which in turn contains functions call of the said contract and the arguments associated. So the symbolic execution of smart contracts should also involve symbolic transaction in which both value and data vuffer are symbolic. These symbolic transacions will be supply to all `Ready` states and, according to the Core Engine specification, this should cause the symbolic execution of one transaction.

3.1.5 ContractFuzzer

In[4], a fuzzing framework for smart contracts was proposed. The paper first defined 7 types of attack and the corresponding test oracle defined by the authors themselves. The architecture for the fuzzer is then laid out. It essentially consist of two parts: an online fuzzing tool and an offline EVM instrumentation tool[10]. The offline part is for the monitoring and extracting information for the later vulberability analysis. The

Online fuzzing process Firstly, the fuzzer will analyze the ABI interface and the bytecode of the contract. ABI basically defined things like how exceptions propagates and how parameters are passed to functions. These things are important as for example the test oracle for exception disorder would likely be interest in how exceptions propagates. This step should extract information such as data types of each arg of ABI functions and the signature of the funtions used in ABI functions.

Secondly, the system will perform ABI signature analysis of smart contracts on the Etherscan and index them by the function signature these contracts support. This is necessary for testing the interaciton of smart contracts. Using the result from these two steps, the tool can generate valid fuzzing inputs under the ABI spec and mutated inputs across the 'trust boundary'[9]. The indexed smart contract will also be used to generate input for ABIs, having contract address as arg.

Finally, the system will start fuzzing the ABI interfaces with the generated input until a predetermined testing time is up.

3.2 Summarize the novelty of each solution

3.2.1 Topic paper

Real-world survey The paper seemed to have conducted a real world survey of what kind of attacks has really been conducted, rather than what's possible or not. Similarly, the defenses adopted in the real-world has also been captured by the study.

Zero-day vulberable This paper has identified a number of zero-day vulnerability and zero-day honeypots. The detection of zero-day vulnerability is rarely using other approach such as code analysis as say for example security pattern in[7] by definition cannot detection zero-day exploits. The authors also observed that the reasons might be some methods cannot perform cross-contract analysis etc[12]

3.2.2 Semantic Framewrok

Instructive definition of security properties Under the rigorous framework provided within, we can now conduct a formalized security analysis of smart contract.

Tool to compile bytecode that is secure by construction It enables the development of tool to verifty the security of the code, from which developers could obtain bytecode that is secure by construction. Besides, efficient statc analysis technique could be developed and its soundness be proved using the semantics provided in this paper.

3.2.3 Securify

The use of Domain Specific Language The Securify language can be modified, providing extentibility and flexibility. The method of construction of higher-level language using something akin to Internal Representaion (IR) was seen in another influential work not reviewed here[5], due to similarity.

3.2.4 Manticore

Highly successful open-source project Often solutions proposed in academic jounnals could not gain the requisite attention in the developer community. This could bring the further development of the solution to a stop. Manticore, however, is a very popular OSS which means the software engineering side of this tool will be benefited from the contribution of software development domain expert.

3.2.5 ContractFuzzer

Fuzzing is an automated testing technique using random data as input to computer program. The paper claimed that the work is the first fuzzing framework for detecting security vulnerability of Ethereum smart contract.

3.3 Adavantages and disadvantages of the solution proposed

3.3.1 Topic paper

Advantages

Library of Adversarial Signature These signature might be able to convert to other kinds of object type and export them for interpolation from other detection mechanism

Disadvantages

Dropping data points The paper has dropped data points due to several reasons including closed-source contracts, false positives of integer overflow case because those are originated from "toy contract" and some definitional issue regarding honeypots.

Exclusion of some attacks The authors considered some attacks has no explicit and measurable losses (in terms of ethers) and therefore did not include them in the scope of the study. Some excluded examples are enumerated: out-of-gas attack and Parity Wallet Freeze.

Manual reasoning in signature generation This involves abductive reasoning, which can be a good thing in itself. However, this would certainly lack completeness, which the authors admitted and claimed this acutally is their ultimate purpose of detecting real-world deployed attack rather than detection of all attacks.

3.3.2 Semantic Framework

Advantages

Library of Adversarial Signature These signature might be able to convert to other kinds of object type and export them for interpolation from other detection mechanism

Disadvantages

Dropping some bugs There are two class of bugs which are dropped by the authors, meaning no security properites have been developed to formulate them. The first class is about runtime error such as Gasless Send and Call stack Limit bugs. The authors claimed that it acutally is possible to formalize these exceptions with small-step semantics. Doing it, however, would not gain any meaningful insight into the security of a smart contract. As for the second class of bugs mainly concerns the particulars of the Solidity semantics which developers just have to be familiar with.

3.3.3 Securify

Advantages

Extensible DSL The security patterns is written in the DSL defined by the authors. The set of patterns could be extended easier by security experts in view of the advances in security research. The language itself could possibly be extended to become more expressive as well.

Contract-specific Patterns The extensibility of the security pattern language allows customization of the patterns. Security auditor could write specific contract pattern, rather than using those generic patterns provided as default by Securify.

Localizing error For instruction patterns violation and compliance, Securify could locate the said instruction specifically. For contract patterns, however, it could only flagged the contract as violation or warning without pinpointing instructions involved in rendering the contract vulnerable.

Disadvantages

Missing patterns Some contracts might match neither compliance nor violation pattern and the framework will issue a warning label instead. It is observed that this is undesirable as the number of warning could be huge and developers tend to ignore large number of warnings, especially when no clear feedbacks are provided

Dropping SHA3 EVM instruction It is unclear as to why and the paper didn't explain clearly. On the other hand, Manticore can support SHA3 EVM instruction, noting it is inherently difficult to execute hash functions symbolically[6]

Assumption of complete reachability The assumption might not be realistic and could severely affect the effectiveness of the compliance and violation notification or producing way too many warning labels.

3.3.4 Manticore

Advantages

Modularity The decoupled approach of the framework allows for creating plugin to create event-based analyses.

Availablity of Python API The API allows customization of analysis with various form of instrumentation. "Programmers implement instrumentation in the form of code instructions that monitor specific components in a system"[10] This essentially allows that users to execute callbacks if a given statee is reached. Callback can then access the corresponding State object, giving complete granular control to users. Python is also a hugely popular and easy to learn language which make the framewokr very accessible to users who seek greater control of the system.

Disadvantages

Scalability The experiment conducted in the paper has a timeout limit of 90 minutes for each contract, indicating Manticore or dynamic analysis in general has scalability issue to some extent.

3.3.5 ContraFuzzer

Advantages

Synergy with other tools In this report, several other techniques are reported: Static code analysis like [7] can produce high number of false positives while Dynamic code analysis like [6] using in combination with fuzzing could used to actually generate the input which caused the reported problem in static code analysis.[1]

Disadvantages

Reliance on Manually defined Test Oracles The vulberability will need to be discovered with a corresponding test oracles , meaning zero-day vulnerabilities will not be detected and careful construction of test oracle is required

3.4 Methodology and results

3.4.1 Topic Paper

Methodology

Preliminary results Authors used their implementation to match against the 420 million transactions from which over 1 billion of rows of trace records were generated. It discovered that Call injections has affected the highest number of contracts. Airdop hunting has the highest number of adversarial transactions.

Three experts Three non-authors domain experts are asked to manaully review a selected collection of unique, non-duplicate contracts. Each expert took 30 hours to evaluate 1272 contracts and achieved 96.78% of agreement rate. The point of having these three experts agree on these unique, non-duplicate, yet similar contracts (as in cases of ERC20 as the authors put it) is not explicitly stated. One apparent reason is that the experts' decision are treated as ground truth to measure the performance of the framework.

3.4.2 Semantic Framework

No experiment was conducted. Instead, the paper compared the vulnerabilities covered in other literature and see if the framework proposed could cover all of them. Two class of bugs were left out and the issue was discussed in the disadvantages session.

3.4.3 Securify

Experiments highlights

Effectiveness in identifying correctness and violations Assuming the security patterns written is correct, the framework performed quite well in most categories which are the seven property defined by the authors. In some cases, `Restricted write` and `Transaction Ordering Dependency: TA`, over 20% of the 24594 contracts had neither compliance nor violation, triggering instead a warning signal. The warning signals requires the users to manually inspect to discern true or false warnings.

Manual inspection results The manual inspection revealed that some property such as TA or Restricted transfer property are hard to prove as both categories have almost 20% of false warnings.

Comparing to other Symbolic Checkers Security outperformed the other two checkers noth in violations and false warnings. The paper, however, only reported violations, true warnings, and false warnings on 4 properties defined by authors.

3.4.4 Manticore

Smart Contract Analysis The paper analyzed 100 Ethereum smart contracts with 90 minutes timout limit. The average coverage of Manticore is

65.64% with a similar median. The mean of the total number of states reach was 207.71, with median of 52, indicating outliers.

3.4.5 ContraFuzzer

The paper used 6991 smart contracts and about 80 hours of fuzzing. The result was compare to the state-of-art tool Oyente (at the time of the writing of ContraFuzzer) over two types of vulnerabilities, Timestamp Dependency and Reentracy. Out of the 6991 contract, 459 contracts were reported to be vulnerable. They also report 100% true positive rate for most of the vulnerability type. It is argued that the high true positive rate should be taken in light of the relatively small number of vulnerability found compared to other studies. A detailed case study for particular class of vulnerabilities where false positives were found was included in the Exerimental Analysis session.

4 Summary

4.1 Limitations

The ultimate limitation of all the solution proposed above is the notion of security in smart contract. What is security? One way to think whether smart contract is inherently insecure, or at least prone to, becuase of the freedom and flexibility to do things? Maybe we can borrow some ideas from software development: Memory management in C is considered to be dangerous in the sense that could set the computer on fire, so we abstrat the process out in some higher-level language such as Python, basically forbidding the users from doing malloc stuff. Could we do something like that in Solidity or maybe a "higher" level language interpreted in Solidity could be developed? Or is it impossible to give a clear boundary on what is secure to do in smart contract and forbid users from doing it.

When to use certain solution In the solutions discussed in this report, there are certainly some solutions better suited at certain setting; Manticore, being an OSS, is suitable for everyday users etc. Semantic framework could be seen as a more "academic" way to approach the problem, as in my opinion, the most suitable way to do so. The rigorous frameworks allows us to understand vulnerabilities in a fundamental level and in a forward deductive manner rather than abductive reasoning backward. Although it seemed to be a steep learning curve to get hold of the framework, one could definitely gain invaluable insight as to what could considered to be security properties. Nonetheless, there is a saying goes "If all you have is hammer, everything becomes a nail". We should be better off with more tool in our repertoire for detections, therefore we might not necessarily need to settle for any one of them.

5 Conclusion

This report has summarized and compared different solution proposed by 5 papers for framework to detect smart contract vulnerability on Ethereum. The problem here is difficult due to various historical or technical reasons. Different frameworks has its own advantages and disadvantages, but most of them suffer from the problem of not having the ground truth to compare against. The notion of security properties seemed to be a way forward to see better progress in the pursue of "Smart Contract Heaven". It is also crucial that we must spend more effort in studying the security analysis of smart contract before we can see wider adoption of the platform.

References

[1] Domagoj Babić, Lorenzo Martignoni, Stephen McCamant, and Dawn Song. Statically-directed dynamic automated test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, page 12–22, New York, NY, USA, 2011. Association for Computing Machinery.

[2] Ting Chen, Xiao song Zhang, Shi ze Guo, Hong yuan Li, and Yue Wu. State of the art: Dynamic symbolic execution for automated test generation. *Future Generation Computer Systems*, 29(7):1758 – 1773, 2013. Including Special sections: Cyber-enabled Distributed Computing for Ubiquitous Cloud and Network Services & Cloud Computing and Scientific Applications — Big Data, Scalable Analytics, and Beyond.

[3] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. A semantic framework for the security analysis of ethereum smart contracts. In Lujo Bauer and Ralf Küsters, editors, *Principles of Security and Trust*, pages 243–269, Cham, 2018. Springer International Publishing.

[4] B. Jiang, Y. Liu, and W. K. Chan. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 259–269, Sep. 2018.

[5] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. Zeus: Analyzing safety of smart contracts. 01 2018.

[6] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1186–1189, Nov 2019.

[7] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 67–82, New York, NY, USA, 2018. Association for Computing Machinery.

[8] Wikipedia contributors. F* (programming language) — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=F*_(programming_language)&oldid=993491836`, 2020. [Online; accessed 23-December-2020].

[9] Wikipedia contributors. Fuzzing — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Fuzzing&oldid=992134129`, 2020. [Online; accessed 23-December-2020].

[10] Wikipedia contributors. Instrumentation (computer programming) — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Instrumentation_(computer_programming)&oldid=994642791, 2020. [Online; accessed 23-December-2020].

[11] Wikipedia contributors. Symbolic execution — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Symbolic_execution&oldid=993761097, 2020. [Online; accessed 23-December-2020].

[12] Shunfan Zhou, Zhemin Yang, Jie Xiang, Yinzhi Cao, Zhemin Yang, and Yuan Zhang. An ever-evolving game: Evaluation of real-world attacks and defenses in ethereum ecosystem. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2793–2810. USENIX Association, August 2020.