DappGuard : Active Monitoring and Defense for Solidity Smart Contracts

Thomas Cook
tomcook@mit.edu

Alex Latham
alatham@mit.edu

Jae Hyung Lee
jaehyung@mit.edu

Abstract

Ethereum's smart contracts present an attractive incentive toward participating in the network. Deploying a smart contract allows a user to run a distributed application (Dapp) that includes storage, payment features, and cryptographic services all within the context of just a contract script and its layout. However, recently exploited vulnerabilities in the Solidity smart contract language have undermined the integrity of Ethereum's smart contract implementations. After some discussion of previous work, we examine whether known vulnerabilities can be detected as attacks post factum from information available on the Ethereum blockchain. Then, we present findings on what information is available for a few selected contracts. Finally, we propose our design for a live monitoring and protection system based on our research findings, the prototypes we developed to gather data, and documented plans for extension.

1 Introduction

Emerging blockchain technologies have shown great promise in offering decentralized services backed by a distributed ledger. This has especially proven true with the deployment of various cryptocurrencies, the most prominent being Bitcoin, Ripple, and Ethereum, all with market capitalization (as of writing) of over $USD 5 billion. The Ethereum blockchain is current the most popular cryptocurrency that utilizes the smart contract abstraction layer to log transactions.

1.1 Smart Contracts and Dapps

Smart contracts allow accounts on the Ethereum network to host, on the blockchain, a 'near' Turing-complete program that supports publicly verifiable transactions. It can facilitate the exchange of money, content, property, shares, or anything of value. When run on the blockchain a smart contract becomes like a self-operating program that automatically executes when specific conditions are met. Because smart contracts run on the blockchain, they run exactly as programmed without any possibility of censorship, downtime, fraud or third party interference. [2]

A majority of smart contracts rely on a traditional web app front end to provide the user interface for that contract's functions. A function is invoked, when a user of the Dapp sends a transaction using his/her account address to the contracts address. For the most part, the functions current smart contracts provide can be categorized as one of financial, notary, library, gaming, or that of a wallet. [6]

1.2 Solidity, Ethereum Virtual Machine, and Ethereum Networks

Solidity is a imperatively typed programming language, similar to Javascript, for writing smart contracts. Solidity smart contracts are compiled to EVM bytecode before being deployed on the blockchain.

Ethereum smart contracts are run on the Ethereum Virtual Machine, which is isolated from any miner or contract owner's local machine, and nodes on the EVM executed smart contracts–

code that executes conditioned transactions between users and other contracts. Contracts can store value, transfer and receive value, and execute code. The execution of transactions with these contracts is performed by nodes on the EVM in a publicly visible way, and paid for in units of *gas*.

The amount of gas allocated for a given transaction's execution is finite and fixed by the creator of the transaction. The gas cost for each EVM instruction is provided in [12] and roughly correlates to the computational cost of each instruction making exceptions where other system goals are considered. For example, arithmetic operations are very cheap while storage is among the most expensive operations so that the blockchain does not grow too quickly.

A smart contract's transaction is executed when it is mined into a block. Miners receive a static reward, as well as the value of any gas used times the gas price (set by the market and having units ether/gas) when they successfully solve a proof-of-work puzzle and submit a block to the blockchain.

1.3 TheDAO Incident

A DAO is fully autonomous, decentralized organization with no single leader. It is designed to replace the rules and structure of a traditional organization, eliminating the need for people and centralized control. "TheDAO" is one kind of DAO initiated and developed by German startup Slock.it. "TheDAO" was launched in April 2016 became as one of the most popular smart contracts since the Ethereum blockchain started running. However, several Ethereum developers raised questions and concerns about its potential vulnerability within its recursive call of splitDAO function which is used for person who want to leave "theDAO" contract. In mid of June 2016, one of "theDAO" creator announced that the "recursive call bug" had been addressed and no "theDAO" contracts and funds were at risk. [1]

However, 3 days later after the announcement, unkown attacker could drain more then 3.6 million ethers by utilizing bug in the recursive spilitDAO function. The Ethereum development team did attempt to split "theDAO" contract to prevent more ether from being taken, but the attempt did not work since they could not get the votes necessary in such a short time. In characteristic of smart contract, developers and designers didn't consider such an attack or malicious usage condition within "theDAO". [4]

1.4 Testing Methodology

The contribution of this paper is twofold. Motivated by existing security analyses of smart contracts in Ethereum, we first demonstrate that there is indeed opportunity to infer attacks from blockchain metadata. Second, we describe and propose future directions for our prototype of DappGuard. Addtionally, we conclude with general discussion of smart contract security as it relates to the Dapp and smart contract environment.

2 Background and Related Work

The Solidity language does not introduce constructs to deal with domain-specific aspects since its computation steps are recorded on a public blockchain. Hence, Ethereum developers had focused on correctness of its execution to avoid manipulation and tamper of adversary. However, in last couple of years it has been proven that correctness of the execution cannot guarantee security of the smart contract not only from reordered. several security vulnerabilities in Ethereum smart contracts have been discovered both by hands-on development experience. [5]

2.1 Security Issues

Most notably, a taxonomy of vulnerabilities and related attacks against Solidity, the EVM, and the blockchain were recently described by Atzei, Bartoletti, and Cimoli [5]. They defined 12 vulnerabilities among these components that pose risks to the contract owners. We utilize this vocabulary in our analyses and thus, we provide the list and brief descriptions for clarity and convenience.

1. **Call to the unknown**: This refers to any use of `call` , `delegatecall` , `send`, or direct call to another contract that results results in execution of an unknown, possibly malicious, fall-

back function (the anonymous function on every smart contract invoked as a catch all).

2. **Gasless send**: If the callee of a `send` (the Ethereum function to transfer ether) is a contract with a relatively expensive fallback function, then the amount of gas the caller is limited to– 2300 units for sending ether to an address– will be insufficient, and an out-of-gas exception will be thrown.

3. **Exception disorder**: refers to unchecked `send` errors or `called` contracts that throw exceptions. The effect is that the calling contract transaction is entirely reverted and all gas is lost.

4. **Type casts**: if the arguments to a direct call from one contract to a function of another contract is incorrectly typed, or the address of the called contract is incorrect, either nothing will happen, or the wrong code will execute. In neither case is an exception thrown, and the caller is not made aware.

5. **Reentrancy**: In some cases, a contract's fallback function allows it to re-enter a caller function before it has terminated. This can result in the repeated execution of functions intended only to be executed once per transaction.

6. **Keeping secrets**: by the public nature of the blockchain, contract fields marked `private` are not guaranteed to remain secret- to set a private field, a contract owner must broadcast a transaction. Cryptographic protocols are required to guarantee that fields are not visible to anyone mining or inspecting the blockchain.

7. **Immutability**: when a contract is added to the blockchain, there is no way to edit it. If a contract is found to be defective, there is often nothing to be done short of killing the contract (assuming measures were taken by the contract creator to make this possible).

8. **Ether lost in transfer**: If ether is sent to an "orphan" address that doesn't actually belong to any user or contract, that ether will be lost and cannot be retrieved.

9. **Transaction Ordering Dependence (TOD)** : This occurs when the assumed state of the blockchain is not the blockchain's actual state when a transaction is executed. The order in which transactions are mined can have adverse effects on the execution of any given transaction. This bug is said to be present in up to %15.8 of all contracts on the blockchain[1].

10. **Stack size limit**: a transaction's call stack grows with each contract invocation, and once the stack is 1,024 frames tall, an exception is thrown. Changes to gas rules and certian instruction costs have resolved this vulnerability in the current enviromnent.

11. **Generating Randomness**: many contracts that require random numbers use the hash of transactions yet-to-appear in the blockchain. A malicious miner could arrange their block to influence the outcome of this random number generation.

12. **Timestamp dependence**: some Dapps use timestamps to generate random numbers. However, the clock in Ethereum is set by the local clocks of its miners. So, such Dapps can be influenced with slight adjustments to miners' reported times.

Additionally, there have been at least 20 reported types of integer orverflow/underflow vulnerabilities. Finally, the solidity documentation maintatins a list of known bugs by version.

2.2 Static Analysis

It was demonstrated that many contracts are likely vulnerable to exception disorder in the form of unchecked 'send' instructions [3]. Further, the evmdis [10] tool performs disassembly of contract bytecode and JUMP target analysis.

2.3 Symbolic Execution: Oyente [11]

Luu and colleagues developed the Oyente symbolic execution system with the security goals of mit-

[1] https://medium.com/@hrishiolickel/why-smart-contracts-fail-undiscovered-bugs-and-what-we-can-do-about-them-119aa2843007

igating TOD, exception disorder, and timestamp dependence bugs for contract owners in the pre-deployment stage and users in the pre-transaction stage. Oyente is able to, given the Ethereum global state and a contract's bytecode, whether the contract's execution has feasible paths to the bug and with what input and path constraints. Oyente's design also utilizes a black box validator to identify and remove false positives.

2.4 Formal Verification

Smart contract security has seen some progress via formally verified proofs about certain contract properties. For example, Bhargavan and showed that by decompiling certain Solidity contracts to F*, bounds can be proven on the gas consumed by a `call` invocation. [7]. The most prominent shortcoming currently is that the system used to translate the Solidity code does not support all Solidity features.

2.5 Best practices

Best practices for writing safe smart contracts are scattered across the Ethereum community. [9, 8]. Many of these are guidelines for engineering secure mechanisms into smart contracts such as fail-safe modes, circuit breakers, and assert guards. These mechanisms have been studied in other contexts, and we adopt some of them into our design where applicable. Other best practices actually eliminate many vulnerabilities. Additionally, these best practices have become Ethereum Improvement Proposals, which introduce lasting changes to the environment. EIP 150, one notable example, essentially solved the stack size limit bug. For example, using name resolution for called contracts (which is provided by some library smart contracts). Further, using blocknumber over timestamp avoids any miner attacks on timestamp dependence. Finally, timed commitments are one way to overcome the keeping secrets bug. [5]

3 Our Contribution

Our work consisted of two phases. First, an investigation of what information about smart contract transactions exists on the Ethereum blockchain, if this information is enough to identify potentially malicious transactions, and finally, if these markers provide reason to think that common vulnerabilities are actually being exploited in selected live contracts. Our findings motivate the second component, DappGuard, a system for live smart contract monitoring.

3.1 Searching for attack fingerprints

For each common vulnerability we identified, we first considered what the visible effects of an exploit of that vulnerability might be. The codification of these side-channel effects informed our subsequent analysis of the live contracts.

Keeping Secrets, TOD This vulnerability is common in 'random' gambling-based contracts, as users might be able to influence the outcome of a contract's execution by broadcasting a large number of transactions immediately preceding the execution. Here, we would expect to see a burst of activity from a given user in a short period of time, which, within a given period, could present as higher-than-average gas costs, or as larger bets, given the influenced outcome (which might mean a higher-than-average transaction value when they place their bet).

Reentrancy Reentrancy will result in multiple calls to a contract's fallback function, which may be easily visible in the offending transaction's log. Here especially, abnormally high gas usage could be a symptom of an attack. Comparing a normal transaction to a malicious transaction exploiting reentrancy on our SimpleDAO contract, the amount of gas used in the malicious transaction was directly proportional to the number of reentrancies that contract was able to make.

Gasless send, call to the unknown These are most often caused by mistakes in writing either transactions or contracts, rather than exploitable vulnerabilities, but they can result in unexpected consequences (i.e. the seemingly unpredictable execution of the recipient contract's fallback function). It is not likely that gasless sends are intentionally provoked by a malicious user, because a

contract would likely stand to gain more value in ether than the gas wasted by a calling address who would get an out of gas exception. Likewise, calls to the unknown are likely mistakes rather than the result of malicious activity, and tools provided by many Ethereum wallets as well as Dapps like the Ethereum Name Service can help prevent erroneous sends.

Integer overflow/underflow Atzei, Bartoletti, and Cimoli identified a second possible of attack on the DAO contract that relies on an integer underflow to allow a malicious contract to withdraw more ether than it deposited. This is a more subtle vulnerability than reentrancy, as it only involves two calls to the malicious contract's fallback, making the attack less obvious and less likely to result in an out-of-gas exception. However, we still expect such an exploit to consume more gas than a normal call, and we might also be able to detect the two fallback invocations in the affected transaction's log.

Exception disorders Exceptions in Solidity are handled inconsistently, with behavior depending on how the inciting call was made. This means that the identification of non-exception side-channel effects of exploits (and exceptions, in any case, aren't always a result of certain exploits) is especially important in identifying exploits that intentionally or unintentionally take advantage of Solidity's confusing exception handling model.

3.2 Testing Methodology

What is likely common between the effects of exploits on common smart contract vulnerabilities are high gas usage, strange message values (especially in transactions on 'random' gambling contracts), and suspicious fallback invocations that might be visible in a transaction's log. We are interested in the prevalence and detectabbility of these effects in real contracts.

3.2.1 SimpleDAO

To test the likelihood of a given contract being subject to malicious behavior, we first deployed a smart contract modeling the DAO contract and attempted to replay the attack against the DAO (an instance of reentrancy). After depositing (test) ether into the contract, called SimpleDAO, on behalf of multiple other contracts, our malicious contract was able to withdraw more ether than was deposited on its behalf. Once we initiated the reentrancy with a call to the malicious contract's fallback (which occurs any time ether is transferred to a contract), the contract was able to perform two withdrals of one ether each (after depositing only one) before running out of gas.

That our example exploit was successful, and required extremely simple techniques on behalf of the malicious party, we expect that the same vulnerability that affected the DAO is still possible and implementable (especially given that many contracts publish their source code publicly). Without attention to the transaction receipts, or if a contract has a particularly large gas allocation, such an attack might go unnoticed, especially on high-value contracts with many transactions. The feasibility of this exploit, and the ease with which it might go undiscovered in less-dramatic instances than of the DAO, is evidence that the attacks we identified are very possible to execute.

3.3 Live Contract Analysis

Seeing that a common smart contract exploit was feasible, and could potentially be detected via aberrant gas usage or message values, we analyzed the transaction receipts for several gambling-based Dapps.

3.3.1 Transaction Receipts

For each of the Dapps EthereumLottery, Etheroll, HonestDice, and Rouleth (versions 3.5 and 4.8), we retrieved transaction receipts for up to 10,000 transactions on the contract. Each transaction receipt is of the following form:

```
{
    "blockNumber":"3702663",
    "timeStamp":"1494720276",
    "hash":"0x662aca...",
    "nonce":"40",
    "blockHash":"0x64b...",
```

```
    "transactionIndex":"12",
    "from":"0xa4fc86...",
    "to":"0x9473bc8b...",
    "value":"200000",
    "gas":"303587",
    "gasPrice":"10000000000",
    "isError":"0",
    "input":"0x727b1...",
    "contractAddress":"",
    "cumulativeGasUsed":"752879",
    "gasUsed":"203587",
    "confirmations":"3612"
}
```


Transaction receipts only refer to transactions that have been successfully added to the blockchain, so `blockNumber` refers to the block where the transaction was mined (and `timeStamp` the time it was mined). The `hash` is the transaction's identifier. `Nonce` is a hash of the proof of work required of the miner who mined the block containing this transaction, and the transaction's *index* refers to its position relative to other transactions on the block. The number of miners who agree on this transaction's outcome is given in `confirmations`. `ContractAddress` is empty for all transactions that do not create a contract (for transactions where the contract of interest is called by another user or contract, the `to` field will be the address of the contract of interest).

For the purposes of our analysis, we are interested in the following fields:

- `from`: This is the address of the calling contract or user. The large number of transactions for some contracts means we might identify patterns in certain users' behavior.
- `isError`: This flag is set to `1` when an exception occurred during the execution of the transaction. Exceptions are reasonably common (and can be the result of a simple mistake in writing contracts or transactions), but if specific users generate many exceptions when interacting with a given contract, we might be suspicious.
- `gasUsed`: This measures (in units of gas) the amount of gas consumed during the transaction's execution. Many exploits come with the side effect of using more gas than we might expect, and so we can check for suspiciously high gas usage among users. Again, gas usage can be variable and doesn't necessarily indicate wrongdoing, but consistently high gas usage from a single user might be a sign of an attacker.
- `value`: We are examining gambling contracts, and so the value (in *wei*, which is equivalent to 10^{-18} ether) often represents a bet. A spike in message value might represent a suspiciously high bet, perhaps indicating that the user has more confidence than others about the outcome of the gamble.

Given this information, we gathered the following data for each contract tested:

- For each user, the number of transactions that resulted in exceptions.
- For each user, the amount of ether transmitted and gas per transaction (and the respective averages per user).
- The total amount of gas spent on the contract's transactions, the total value transferred, and the total number of transactions considered.
- The average and standard deviation of gas usage over all of the contract's transactions, and the average and standard deviation of value over all of the contract's transactions.
- For each user and each transaction, the number of standard deviations away from the contract average for gas and value the transaction's gas usage and value lie.
- The number of users who had transactions where gas expenditure fell outside two standard deviations of the contract's average.

Of course, evaluating value and gas usage across users is less illustrative if gas expenditure and message values seem random. However, as we would expect, in the contracts we tested gas usage tends to peak around certain values (presumably, functions in the called contract expend reasonably consistent amounts of gas per call), and values seem similarly

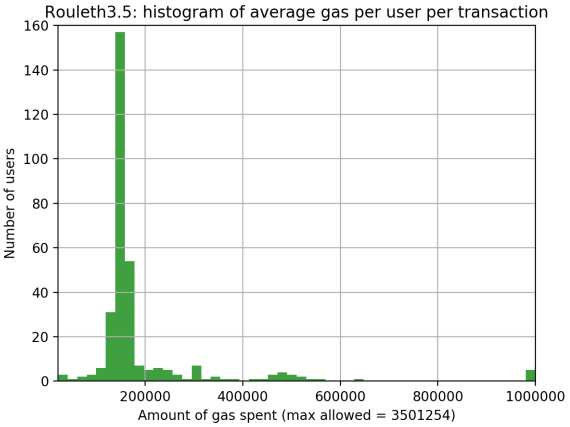
clustered. The standard deviation calculations assumed a roughly normal distribution for gas and value, but this assumption was likely inappropriate for some of the contracts. More refined outlier detection would be necessary to evaluate contracts with more complicated distributions.

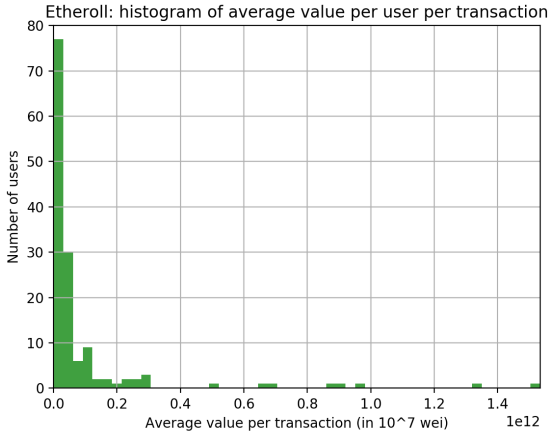
3.3.2 Observations

From this analysis, we noticed the following[2]. First, we observed several users with high proportions of erroneous transactions– one user of Rouleth3.5 (the largest contract we tested by number of transactions) had a total of eight transactions with the contract, seven of which resulted in exceptions. We also saw a number of users who had at least one transaction with gas usage more than two standard deviations above the contract's average (for Rouleth3.5, 92 such unique addresses were identified from the most recent 10,000 transactions).

Second, we found instances of contracts with well-grouped distributions of average gas use and value transferred per user. Histograms of average gas usage per user reveal that some contracts admit to normal-looking distributions of average user gas expenditure per transaction. Pictured is such a histogram for Rouleth3.5, where we can clearly see a number of users that, on average, used significantly more gas than most– but, according to the amount of gas allowed, did *not*, on average, run out of gas. This result is of considerable interest- instances of gas usages that are much higher than average could definitely be suspect– but, because the gas used was still within the amount allocated, it caused no visible effects. Of course, there are legitimate reasons why a transaction might use a lot of gas, but this is to show that malicious executions that will definitely use excess gas can be identified.

An example of a histogram of average user value per transaction is shown for Etheroll. Here, we see that a large proportion of transactions transferred a small amount of Ether, but a handful transferred much more– on the order of ten-times more, in some cases. Again, a high value transfer is not certainly suspect, but this is to show that outliers, at least for some contracts, can be identified.





4 Proposed Design

4.1 Threat Model

DappGuard assumes that an attacker holds at least one Ethereum account and may or may not also be a miner. We also assume the contract creator follows best practices and deploys formally verified contract.

4.2 System and Goals

In order to collect data from the blockchain, one needs to run a blockchain node with one of the available clients. The most popular clients for Ethereum are geth written in Go and parity written in Rust. These clients have managment APIs that allow ex-

[2] All transactions initiated by the contract creator were omitted from this analysis. Contract creators might call a host of functions on their own contract that do not represent standard use of the contract, and the contract owners often make a large number of transactions that would likely have skewed our analysis.

ecution tracing and In our experimentation, to deploy test contracts without needing to spend real monetary value holding ether, we used a testnet and experimented implemnting a fail-safe by sending transactions to our contract. As a result, we present a design for DappGuard.

The goals of DappGuard are relatively simple:

1. Classify known attacks from transaction data
2. Protect smart contracts from known attacks
3. Determine malicious actors and learn new attacks

To accomplish these goals, DappGuard's architecture relies on the following components:

- EthWorldModel (EWM)
  - ContractModel
  - AddressModel
  - MinerModel
  - Rules
- Blockchain Monitor Nodes (BMN)
  - Mainnet node
  - Private testnet Clone node
- TransactionRiskAnalyzer (TRA)
  - Oyente Validator
  - Transaction Foretrace
  - Anomaly Analysis + Rules Engine
  - Responder

The EWM stores information about transaction histories for the contract, for users who have interacted with the contract, and metadata associated with the blocks containing each transaction. The ContractModel stores statistics relevant to the classification of any new transactions– such as distributions of gas usage and value transferred from past transactions and the number of transactions on the contract (in total, and from particular users). These models will resemble a refined version of our live contract analysis (Sec. 3.3).

The AddressModel represents each address's interaction with the contract– amount of value transferred, average gas usage, and the number of exceptions among its transactions with the contract.

The MinerModel serves to relate transactions on the contract wiht the details of the blocks and miners responsible for adding them to the blockchain. Such information as who mined a block containing a transaction on the contract, the order of transactions on that block, associated with information from the AddressModel for the source of the transaction.

The Rules Engine works to apply each model to new transactions to inform judgments about the validity of the transaction, and updates each model to strengthen the identification of standard and malicious transactions and mining patterns.

The BMNs serve to keep an up-to-date version of the blockchain available (the Mainnet node), and to test the execution of pending transactions on our contract (the Testnet node). These nodes are each Ethereum nodes on the main network and a test network (such as Ropsten or Kovan) respectively. The Mainnet node's job is to remain synced with the main blockchain. The job of the Testnet node is to trace the possible states for all orderings of transactions on our contract that are pending on the main network.

To identify security risks associated with TOD, exception disorders, and timestamp dependencies, we validate pending transactions against the results of the Oyente symbolic execution system. This validator uses the results of running Oyente on our contract, in the form of constraints on function inputs that produce bugs of these kinds, to determine the validity of the transaction's execution. Security issues outside the scope of this validator will be addressed by either the Testnet simulation or application of our Rules Engine.

Finally, the Responder module reacts to dangerous transactions as determined by the contract. This might send a notification to the contract owner (as a logged event on the Testnet node), or initiate a takedown of the contract, or kick off a transaction generated by the user that changes the contract's state to prevent an imminent transaction from exploiting a security hole.

4.3 Modes of Operation

4.3.1 Knowledge Acquisition

This mode refers to DappGuard initially populating, maintaining, and optimizing its EthWorldModel for efficient transaction analysis. This would include gathering a complete blockchain transaction history and analysis, and updates to any known vulnerabilities or risk indicators used by the Rules Engine. Any post-mordem analysis of missed attacks would be one example. Finally, any information exchange with client smart contract owners that results in changes to the EWM would fall under knowledge acquisition for DappGuard.

4.3.2 Active Monitoring & Detection

For any client smart contracts that DappGuard manages, it actively monitors the incoming transactions. For all incoming transactions to the contract of interest, we the use the saved result of Oyente excution for the contract to determine quickly whether the contract will experience a TOD, exception disorder, or timestamp dependence bug. Then, for remaining potential vulnerabilities, the contract execution is simulated on the private testnet and any attacks on that execution are detected using the Anomaly Analysis and Rules Engine. Finally, if a threat is detected that will possibly transpire, DappGuard will send a transaction, that with high probability, gets mined before the risky transaction.

5 Suggestions for Future work

As to future work, we mention directions we considered before finalizing our decision. Although Solidity was the only smart contract language we considered, others too have emerged such as the functionally typed Bamboo, which hopes to mitigate immutability bugs. Additionally, we considered that because Dapps have traditional web front ends, that they could be vulnerable, too, to traditional web exploits. The last attack vector we considered was that of padding in the Ethereum Virtual Machine and malleability of transactions. This is protected currently, so long as the `keccak256` hash of the data is verified and that `keccak256` remains secure.

6 Discussion

Obviously, the current incarnation of DappGuard is little more than proof-of-concept based on our developed tools. However, we believe that our research shows that blockchain data and knowledge of smart contract attack mechanics can be used to detect attacks as they occur. Further, we believe by providing this design, we both motivate active and adaptive security in the Ethereum ecosystem and identify strong candidates for tools a system of this nature would rely on.

References

[1] Ethereum Hack dao hack simplified. http://blog.erratasec.com/2016/06/etheriumdao-hack-similfied.html. Accessed: 2016-06-18.

[2] Ethereum what is ethereum for beginners. https://blockgeeks.com/guides/what-is-ethereum. Accessed: 2012-01-30.

[3] Hacking, Distributed: Scanning live Ethereum contracts for the "unchecked-send" bug, author= Wen, Zikai Alex and Miller, Andrew, year = 2016, howpublished = ://hackingdistributed.com/2016/06/16/scanning-live-ethereum-contracts-for-bugs/.

[4] thedao hack faq. https://ethereum.stackexchange.com/questions/6183/thedao-hack-faq-how-did-the-attack-happen-on-17-june-2016. Accessed: 2016-06-17.

[5] ATZEI, N., BARTOLETTI, M., AND CIMOLI, T. A survey of attacks on ethereum smart contracts. Tech. rep.

[6] BARTOLETTI, M., AND POMPIANU, L. An empirical analysis of smart contracts: platforms, applications, and design patterns. *CoRR abs/1703.06322* (2017).

[7] BHARGAVAN, K., DELIGNAT-LAVAUD, A., FOURNET, C., GOLLAMUDI, A., GONTHIER, G., KOBEISSI, N., KULATOVA, N., RASTOGI, A., SIBUT-PINOTE, T., SWAMY, N., AND ZANELLA-BÉGUELIN, S. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security* (New York, NY, USA, 2016), PLAS '16, ACM, pp. 91–96.

[8] CONSENSYS. Smart contract best practices. https://github.com/ConsenSys/smart-contract-best-practices#eng-techniques, 2017.

[9] ETHEREUM. Security considerations. http://solidity.readthedocs.io/en/develop/security-considerations.html, 2015–2017.

[10] JOHNSON, N. evmdis. https://github.com/Arachnid/evmdis, 2016–2017.

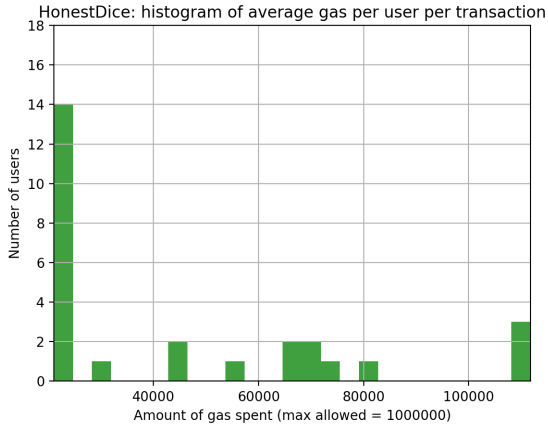
[11] LUU, L., CHU, D.-H., OLICKEL, H., SAXENA, P., AND HOBOR, A. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2016), CCS '16, ACM, pp. 254–269.

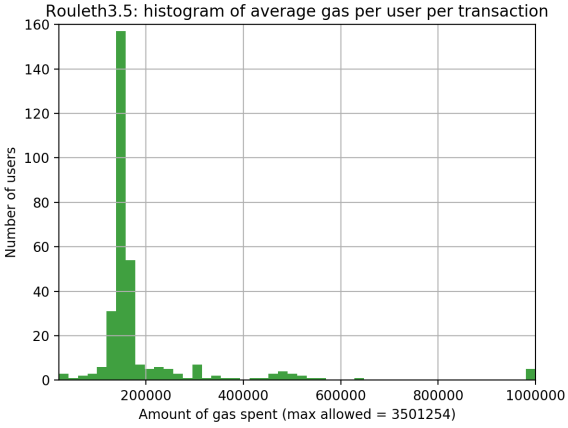
[12] WOOD, G. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper 151* (2014).

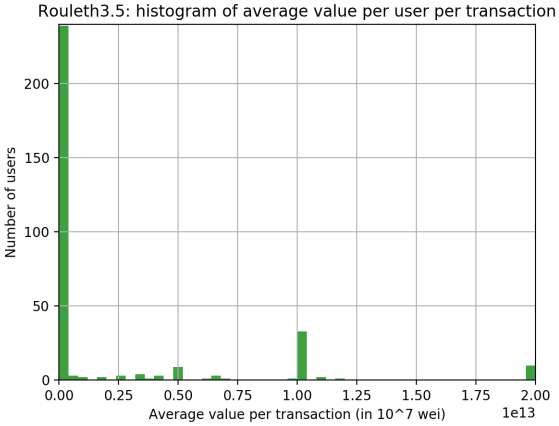
Code

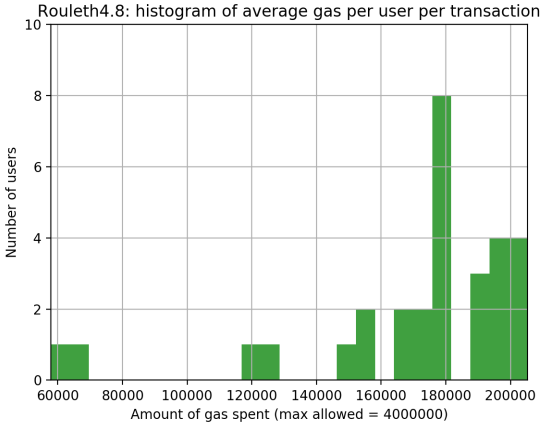
All code relevant to this project can be found in the following repository: https://github.com/cookt/857final

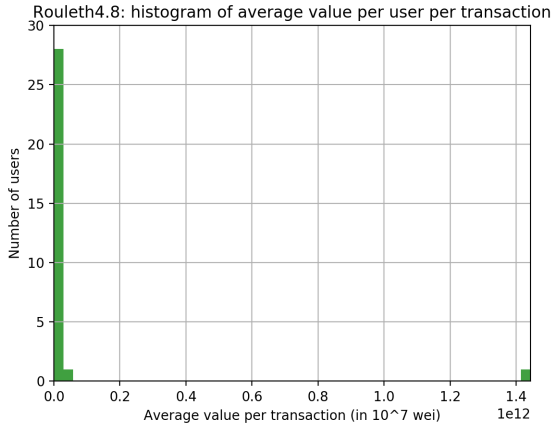
Appendix of Live Contract Gas and Value Histograms

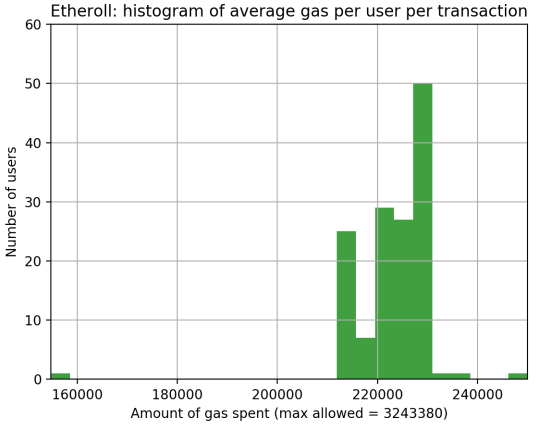


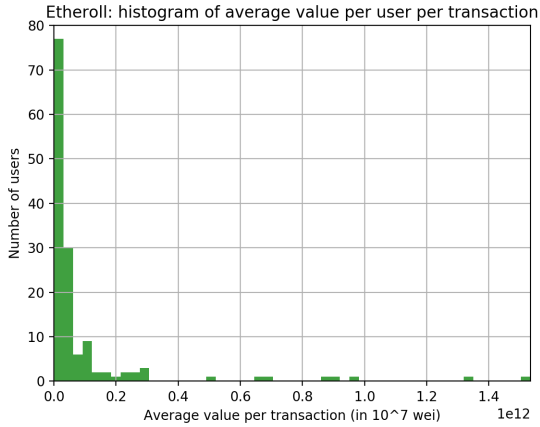


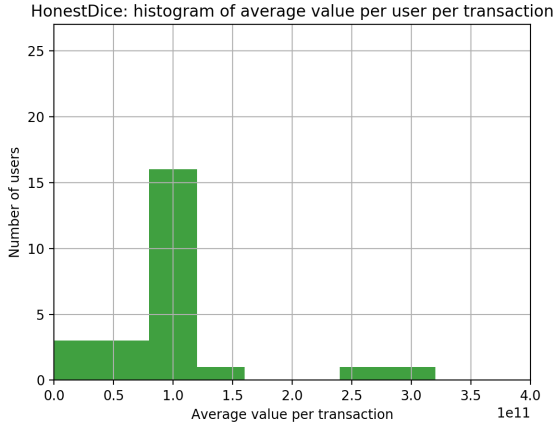









Etheroll: histogram of average value per user per transaction
Number of users
Average value per transaction (in 10^7 wei)
1e12


HonestDice: histogram of average value per user per transaction
Number of users
Average value per transaction
1e11