

Under-Optimized Smart Contracts Devour Your Money

Ting Chen^{*†}, Xiaoqi Li[†], Xiapu Luo^{†‡}, Xiaosong Zhang^{*}

^{*}Center for Cybersecurity, University of Electronic Science and Technology of China, China

[†]Department of Computing, The Hong Kong Polytechnic University, China

Email: brokendragon@uestc.edu.cn, csxqli@gmail.com, csxluo@comp.polyu.edu.hk, johnsonzxs@uestc.edu.cn

Abstract—Smart contracts are full-fledged programs that run on blockchains (e.g., Ethereum, one of the most popular blockchains). In Ethereum, gas (in Ether, a cryptographic currency like Bitcoin) is the execution fee compensating the computing resources of miners for running smart contracts. However, we find that under-optimized smart contracts cost more gas than necessary, and therefore the creators or users will be overcharged. In this work, we conduct the *first* investigation on Solidity, the recommended compiler, and reveal that it fails to optimize gas-costly programming patterns. In particular, we identify 7 gas-costly patterns and group them to 2 categories. Then, we propose and develop *GASPER*, a new tool for automatically locating gas-costly patterns by analyzing smart contracts' bytecodes. The preliminary results on discovering 3 representative patterns from 4,240 real smart contracts show that 93.5%, 90.1% and 80% contracts suffer from these 3 patterns, respectively.

I. INTRODUCTION

The success of Bitcoin, a decentralised cryptographic currency that reached a capitalisation of 10 billions of dollars since its launch in 2009 [1], has attracted lots of attentions from both industry and academia to investigate the underlying technology of cryptocurrencies, the blockchain. One prominent application on blockchains is to execute smart contracts, which can be considered as full-fledged programs running on blockchains from the perspective of software engineering.

Ethereum is one of the most popular blockchains where more than 10 million transactions had occurred [2]. The term “blockchain” and “smart contract” refer to the Ethereum blockchain and its smart contracts, respectively, below without special declaration. A smart contract can be developed in Solidity (the recommended language), Serpent, or LLL. No matter which programming language is used, the source of a smart contract will be compiled into bytecodes that can be executed in the Ethereum Virtual Machine (EVM for short).

Smart contracts run on the machines of miners, who can earn Ethers (i.e., the cryptographic currency circulated in Ethereum) by contributing their computing resources. The creators and users of smart contracts will be charged certain amount of gas for purchasing the computing resources from miners. The charge of a transaction equals to the multiplication of the gas consumed by executing the transaction and the price of gas (Ether per unit). Moreover, when deploying contracts, the creators will also be charged of gas, the amount of which are related to the size of smart contracts in bytecodes.

We find that under-optimized smart contracts cost more gas than necessary, and therefore the creators or users will be overcharged. To save money, developers had better follow gas-efficient programming patterns. Unfortunately, there is not such a guideline yet, and it is difficult for developers to identify gas-costly bytecode and replace them with gas-efficient ones, because it requires deep understanding of EVM's instructions, the gas consumption for different operations, the data locations accessed by operations, the amount of data read or written etc. Hence, a compiler that can optimize the bytecode for minimizing gas consumption is highly desired.

In this paper, we conduct the *first* investigation on Solidity, the recommended compiler for Ethereum, and reveal that it fails to optimize gas-costly programming patterns. More precisely, we identify 7 gas-costly patterns and divide them into 2 categories: useless-code related patterns, and loop-related patterns. Furthermore, we propose and develop *GASPER* (short for GAS-costly Patterns checker), a new tool for discovering gas-costly patterns in bytecode automatically. *GASPER* leverages symbolic execution and it currently can locate 3 representative patterns, which cover the two categories. By applying *GASPER* to analyze all deployed smart contracts until Nov. 5th, 2016, we find that 93.5%, 90.1% and 80% smart contracts suffer from these 3 patterns, respectively. It is worth noting that although the list of our patterns is by no means of complete, this research sheds light on this important issue and hopefully stirs more research on it.

Overall, we make the following contributions:

- To our best knowledge, this is the *first* investigation revealing that lots of smart contracts, generated by the recommended compiler, contain gas-costly bytecodes, which can be replaced with gas-efficient bytecodes to save money.
- We propose and develop *GASPER*, a new tool based on symbolic execution for automatically discovering gas-costly patterns in bytecode. The current version covers 3 representative patterns in 2 categories, and is being extended to support more patterns.
- We apply *GASPER* to all deployed smart contracts until Nov. 5th, 2016, and find that 93.5%, 90.1% and 80% smart contracts suffer from these 3 patterns, respectively.

[‡] The corresponding author.

II. BACKGROUND

Gas is used for purchasing computing resources from miners since smart contracts run on miners' machines. Gas can be considered as money with equivalent value. For example, the average gas price on Nov. 11, 2016 is 0.000000024334480804 Ether [3], which is roughly equal to 2.5×10^{-7} US dollars [4]. Note that the gas price and the exchange rate of Ether to US dollar are determined by the market and keep changing.

Deploying and executing smart contracts cost money. For instance, an addition operation that sums up the top two items of the stack takes 3 units of gas, about 7.5×10^{-7} US dollars. One may argue that the cost for an addition is so low that we do not need to optimize it. However, it is worth noting that real smart contracts consist of lots of operations and some operations consume much more gas than the addition operation, as shown in Table I. Moreover, smart contracts usually provide public methods that can be called unlimited times by various clients and contracts. Hence, an optimized smart contract can save obvious gas (i.e., money) than its unoptimized counterpart due to the scale effect.

TABLE I: Gas cost of different operations, a complete list can be found in Ethereum's yellow paper [5]

Operation	Gas	Description
ADD/SUB	3	Arithmetic operation
MUL/DIV	5	
ADDMOD/MULMOD	8	
AND/OR/XOR	3	Bitwise logic operation
LT/GT/SLT/SGT/EQ	3	Comparison operation
POP	2	Stack operation
PUSH/DUP/SWAP	3	
MLOAD/MSTORE	3	Memory operation
JUMP	8	Unconditional jump
JUMPI	10	Conditional jump
SLOAD	200	Storage operation
SSTORE	5,000/ 20,000	
BALANCE	400	Get balance of an account
CREATE	32,000	Create a new account using CREATE
CALL	25,000	Create a new account using CALL

Stack operations (e.g., POP, PUSH), arithmetic operations (e.g., ADD, SUB), bitwise operations (e.g., OR, XOR), and comparison operations (e.g., LT/GT) are cheap because being a stack-based virtual machine, EVM favors such stack-related operations. Loading a word (i.e., 256 bits) from the memory (e.g., MLOAD) or saving a word to the memory (e.g., MSTORE) are also cheap. The term "memory" referred in Ethereum stands for a special memory area, of which a contract obtains a freshly cleared instance for each message call. For example, the data attached in a message call is stored in memory. It is worth noting that the gas consumption will be multiplied if many words in memory are read or written. Moreover, memory can be expanded when accessing a previously untouched memory location. Every expanded word needs 3 units of gas.

Loading a word from the storage (i.e., SLOAD) or saving a word to the storage (i.e., SSTORE) are expensive. The term "storage" referred in Ethereum is a persistent memory area where any changes to the storage by one call of a contract can be observed by subsequent calls of that contract. A SSTORE operation costs 20,000 units of gas if the storage word is set to non-zero from zero; otherwise, it costs 5,000. It is worth

noting that although the caller of a contract will be refunded 15,000 units of gas if a SSTORE operation sets a non-zero storage word to zero, the refund will not be committed until the transaction completes successfully.

EVM has a number of blockchain-specific operations which are very expensive, such as BALANCE, CREATE and CALL. Moreover, a conditional jump (i.e., JUMPI) is more expensive than an unconditional jump (i.e., JUMP). The gas consumption of each operation is susceptible to change due to the fast evolving of Ethereum. Roughly speaking, users are charged proportionally to the consumed computing resources.

III. GAS-COSTLY PROGRAMMING PATTERNS

We identify 7 gas-costly patterns, which can be classified into two categories: useless code related patterns and loop related patterns. The former introduces additional cost due to the increased size of bytecode during the deployment and the removable bytecode in runtime. The latter involves using expensive operations in the loop. We have validated all these patterns using the latest Solidity (V 0.4.4) whose optimization is enabled. More precisely, we feed Solidity the gas-costly patterns in source code, and then check whether the gas-costly patterns are converted into gas-efficient ones in the generated bytecode. The results show that *none* of these patterns has been optimized by Solidity. For the ease of illustration, we present the patterns in source code rather than bytecode.

A. Category 1: Useless Code Related Patterns

Pattern 1	1 function p1 (uint x){ 2 if (x > 5) 3 if (x*x < 20) 4 XXX }	Pattern 2	1 function p2 (uint x){ 2 if (x > 5) 3 if (x > 1) 4 XXX }
-----------	---	-----------	--

Fig. 1: Pattern 1: dead code, and Pattern 2: opaque predicate

1) Dead code. Fig.1 (Pattern 1) gives an example of dead code where Line 4 will not be executed because the predicate "x*x<20" at Line 3 is evaluated to false under all circumstances. Solidity does not remove Line 3 and 4 from the generated bytecode and hence wastes money.

2) Opaque predicate. The outcome of an opaque predicate is known to be true or false without execution. For example, the predicate "x>1" in Fig.1 (Pattern 2) is an opaque predicate. Since the predicate at Line 3 is evaluated to true under all circumstances, it should be removed for saving gas.

B. Category 2: Loop Related Patterns

Pattern 3	1 uint sum = 0; 2 function p3 (uint x){ 3 for (uint i = 0; i < x; i++) 4 sum += i; }	Pattern 4	1 function p4 () returns (uint){ 2 uint sum = 0; 3 for (uint i = 1; i <= 100; i++) 4 sum += i; 5 return sum; }
-----------	---	-----------	--

Fig. 2: Pattern 3: expensive operations in a loop, and Pattern 4: constant outcome of a loop

1) Expensive operations in a loop. The expensive operations in a loop are worth attention because they may execute multiple times in one invocation. Moving the expensive operations out of the loop can save gas. For example, in Fig.2 (Pattern 3), since the variable *sum* is stored in the storage, Line 4 involves a SLOAD for loading *sum* to the stack and a SSTORE for

saving the outcome of the ADD to the storage. Note that the storage-related operations are very expensive.

An advanced compiler should assign *sum* to a local variable (e.g., *tmp*) that resides in the stack, then add *i* to *tmp* inside the loop, and finally assign *tmp* to *sum* after the loop. Such optimization reduces the storage-related operations from $2x$ to just 2, i.e., one SLOAD and one SSTORE.

2) Constant outcome of a loop. In some cases, the outcome of a loop may be a constant that can be inferred in compilation. As shown in Fig.2 (Pattern 4), the storage variable *sum* in *p4* equals to 5050 after the loop. Hence, the body of *p4* should be simplified as “return 5050;”.

3) Loop fusion. It combines several loops into one if possible and thus reduces the size of bytecode. In particular, it can reduce the amount of operations, such as conditional jumps and comparison, etc., at the entry points of loops. The two loops shown in Fig.3 (Pattern 5) can be combined into one loop, where both *m* and *v* get updated.

Pattern 5	1	function p5 (uint x){	Pattern 6	1	uint x = 1;
	2	uint m = 0;		2	uint y = 2;
	3	uint v = 0;		3	function p6 (uint k){
	4	for (uint i = 0 ; i < x ; i++)		4	uint sum = 0;
	5	m += i;		5	for (uint i = 1 ; i <= k ; i++)
	6	for (uint j = 0 ; j < x ; j++)		6	sum = sum + x + y; }
	7	v += j; }			

Fig. 3: Pattern 5: loop fusion, and Pattern 6: repeated computations in a loop

4) Repeated computations in a loop. In some cases, there may be expressions that produce the same outcome in each iteration of a loop. Hence, the gas can be saved by computing the outcome once and then reusing the value instead of recomputing it in subsequent iterations, especially, for the expressions involving expensive operands. For example, in Fig.3 (Pattern 6), the gas consumption is very high due to the repeated computations. More precisely, the summation of two storage words (i.e., “x+y” at Line 6) is quite expensive because *x* and *y* should be loaded into the stack (i.e., SLOAD) before addition. To save gas, this summation should be finished before the loop, and then the result is reused within the loop.

Pattern 7	1	function p7 (uint x , uint y) returns (uint){
	2	for (int i = 0 ; i < 100 ; i++)
	3	if (x > 0) y+=x;
	4	return y; }

Fig. 4: Pattern 7: Comparison with unilateral outcome in a loop

5) Comparison with unilateral outcome in a loop. It means that a comparison is executed in each iteration of a loop but the result of the comparison is the same even if it cannot be determined in compilation (i.e., not an opaque predicate). For instance, in Fig.4, the comparison at Line 3 should be moved to the place before the loop.

Summary: Adequate optimizations can reduce the cost of contract creators if the size of smart contracts can be reduced (e.g., eliminating dead code, removing unnecessary comparisons), and the cost of contract users if the computations of smart contracts can be reduced (e.g., moving expensive operations out of a loop). It is worth noting that the loop-

related patterns will cost more gas with the increase of the loop count.

IV. GASPER

We propose and develop GASPER to automatically discover gas-costly programming patterns from the bytecode of smart contracts. GASPER handles bytecode directly without the need of source code, because only a few (728 until Nov. 29th, 2016) smart contracts open their sources. As an early research achievement, the current version of GASPER can find all patterns in category 1 and one representative pattern (i.e., expensive operations in a loop) in category 2. The detection of other patterns is in development.

GASPER conducts symbolic execution on bytecode to cover all reachable code blocks (a block is a straight-line code sequence with no branches in except to the entry and no branches out except at the exit). Given a smart contract, GASPER first disassembles its bytecode using *disasm* provided by Ethereum. Then, GASPER constructs the Control Flow Graph (CFG). It is worth noting that the CFG will be improved gradually during symbolic execution if new control flow transfers are found. Symbolic execution starts from the root node of the CFG, and traverses the CFG. If GASPER encounters a conditional jump, it checks which branches (i.e., true or false) are feasible by querying the Z3 solver [6]. If both are feasible, GASPER selects one branch following the depth-first search.

A. Detection of Dead Code

GASPER detects dead code through three steps. First, it logs the addresses of all executed blocks by symbolic execution. Then, it collects the addresses of all blocks by scanning the CFG. Finally, GASPER reports all blocks that are found in the CFG but not executed by symbolic execution as dead code.

B. Detection of Opaque Predicates

To detect opaque predicates, GASPER executes the smart contract symbolically, and records the executed branch (i.e., true or false) when a conditional jump is encountered. After that, the conditional jump with one never-executed branch is regarded as an opaque predicate.

C. Detection of Expensive Operations in a Loop

GASPER detects this pattern through two steps. First, GASPER looks for loops in the bytecode. Second, it searches loop bodies for expensive operations. More precisely, GASPER firstly searches for back edges in the CFG, which indicate the existence of loops, and then identifies the entry block and exit block for each loop. Afterwards, using Dijkstra algorithm, GASPER calculates the distances between each block with the entry block and exit block, respectively. The distance between two nodes is the least number of edges from one node to the other. A block is considered to be in a loop if it is closer to the exit block than to the entry block. Currently, GASPER supports detecting 3 expensive operations, including SLOAD, SSTORE and BALANCE. More operations will be included in future work.

V. EVALUATION

We have implemented GASPER based on OYENTE [7], and evaluated it using all smart contracts deployed on Ethereum. More precisely, we scan all addresses in the blockchain because each deployed contract must be associated with a unique address. We find 566,907 addresses till November 5th, 2016, of which 539,617 addresses contain no bytecodes. Therefore, we download 27,290 contracts' bytecodes in total. Moreover, we find that many contracts are exactly the same (i.e., their bytecodes are identical). After eliminating identical contracts, 4,669 contracts are left. During experiments, 429 (less than 10%) contracts cannot be examined because OYENTE crashes due to its internal errors(e.g., *Unknown Instructiondelegatecall*, *Stack Underflow*, *Unknown Instructionnextcodesize*) or OYENTE runs out of time. Eventually, 4,240 contracts are successfully inspected.

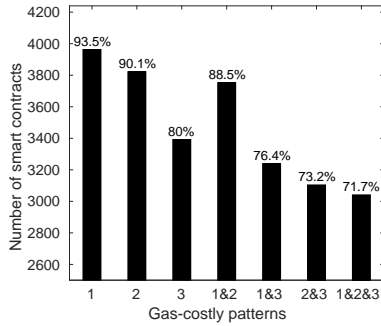


Fig. 5: Overview of gas-costly patterns: 1, 2, 3 indicate dead code, opaque predicates, and expensive operations in a loop, respectively.

The number of smart contracts that have the 3 gas-costly patterns are illustrated in Fig.5. More than 70% contracts contain all these patterns, indicating that their bytecodes have not been properly optimized for reducing gas. Besides, more than 90% contracts have dead code or opaque predicates.

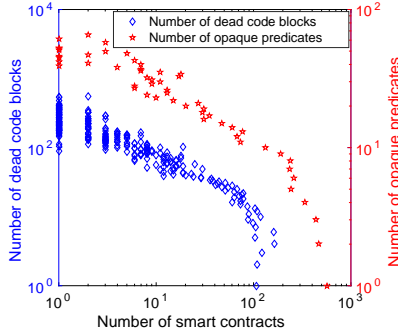


Fig. 6: Distribution of dead code blocks and opaque predicates in smart contracts.

Fig.6 presents the distribution of dead code blocks and opaque predicates in smart contracts. Each point (a, b) indicates that a smart contracts contain b dead code blocks or opaque predicates. Note that the contracts without these two patterns are not counted. The distributions of dead code blocks and opaque predicates demonstrate similar trends: 51.7% contracts contain more than 20 dead code blocks and 52.6% contracts contain more than 10 opaque predicates.

Fig.7 demonstrates that 69.9%, 78.5% and 21% contracts have SLOAD, SSTORE and BALANCE operations in a loop, respectively. Moreover, if a contract has SSTORE operations in a loop (the percentage is 69.9%), it may contain SLOAD operations (69.3%) as well. Interestingly, if a contract uses BALANCE operations in a loop (21%), it likely contains both SLOAD and SSTORE operations (18.6%).

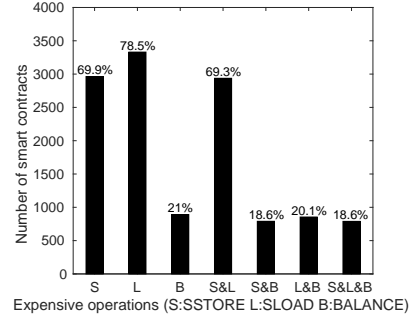


Fig. 7: Number of contracts containing expensive operations

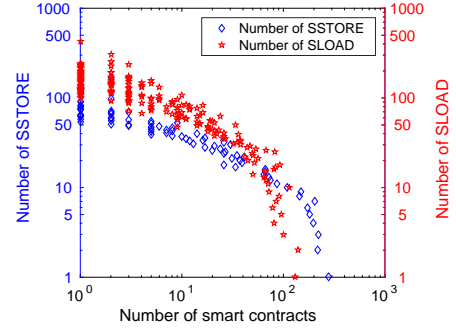


Fig. 8: Distribution of SSTORE and SLOAD within a loop in smart contracts.

Fig.8 shows that a large number of contracts contain many expensive operations in a loop. For example, 57.1% and 51.5% of contracts have more than 7 SSTORE and 20 SLOAD operations in a loop, respectively. Note that contracts without such expensive operations in a loop are not counted.

As expected, contracts with larger size are likely to contain more gas-costly patterns. Fig.9 shows the relationship between the number of SLOAD/SSTORE and the size of smart contracts. For example, a contract, named ARK, which is of 34,767 bytes and deployed in 0x37b4869e73B7cE1284D6502B01aC81d500b50237, has 304 SLOAD and 168 SSTORE operations in loops.

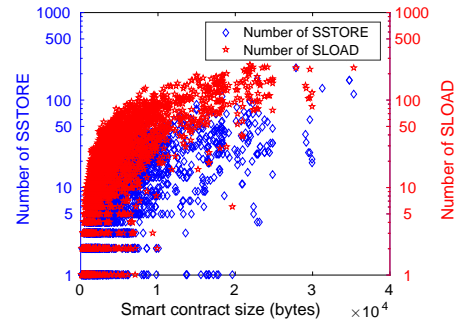


Fig. 9: Statistics of the size of contracts which contain SSTORE and SLOAD in a loop.


```

193 function indexOf (string _haystack, string _needle) internal returns (int)
194 {
195     bytes memory h = bytes (_haystack);
196     bytes memory n = bytes (_needle);
197     if (h.length < 1 || n.length < 1 || (n.length > h.length))
198         return -1;
199     else if (h.length > (2 ** 128 - 1))
200         return -1;
...

```

Fig. 10: Gas-costly code in *FirstContract*

A. Real Case 1: *FirstContract*

FirstContract is open source and deployed at the address 0x68C7147205A8bEB9D99fD19908b93462CdFfC60d. GASPER discovers dead code at Line 200 (i.e., pattern 1) and an opaque predicate (i.e., pattern 2) at Line 199, as shown in Fig.10. The function *indexOf* takes in two strings, *_haystack* and *_needle*. At Line 195, *_haystack* is converted into a set of bytes, *h*. At Line 199, the length of *h* is compared to $2^{128} - 1$. However, the predicate will never be evaluated to true because “**” stands for exponential arithmetic. Consequently, the code at Line 200 cannot be executed.

B. Real Case 2: *Ballot*

```

29 Proposal[] public proposals;
...
57 function winningProposal() constant returns (uint8 winningProposal){
58     uint256 winningVoteCount = 0;
59     for (uint8 proposal = 0; proposal < proposals.length; proposal++)
...

```

Fig. 11: Gas-costly code in *Ballot*

Ballot is also open source and deployed at the address 0x5A4964bb5FDd3CE646bB6AA020704F7D4db79302. GASPER finds a SLOAD operation in a loop and it can be moved outside the loop, as shown in Fig.11.

Since the array *proposals* (defined Line 29) is in the storage, getting access to its length (i.e., *proposals.length* at Line 59) involves the SLOAD operation. Moreover, the number of executing SLOAD is *proposals.length*, because the length of *proposals* is accessed in each iteration of the loop. This costly code can be optimized by assigning *proposals.length* to a stack variable, and then using the stack variable to do the comparison with *proposal* at Line 59. After optimization, the number of using SLOAD can be reduced to only one.

VI. RELATED WORK

There are a few studies on blockchain and smart contracts, but none of them investigates the gas consumption from the same viewpoint as ours. Luu et al. develop OYENTE [7], a novel symbolic execution based tool, to discover security bugs in Ethereum smart contracts. Bhargavan et al. use formal verification to analyze smart contracts (e.g., whether contracts check the return value of a *send* operation because *send* may fail) [8], [9]. HAWK [10] is a decentralized smart contract system enabling developers to write privacy-reserved smart contracts. Juels et al. find that smart contracts can facilitate crimes [11] and show how criminal smart contracts can facilitate leakage of confidential information, theft of cryptographic keys, and various real world crimes. TOWN CRIER [12] aims

to provide trustworthy data to smart contracts because many applications of smart contracts need data from outside the blockchain. Atzei et al. survey a series of attacks which exploit the vulnerabilities of contracts to steal or tamper the assets [13].

VII. CONCLUSION AND FUTURE WORKS

We perform the *first* investigation to expose that lots of smart contracts, generated by the recommended compiler Solidity, contain gas-costly bytecodes, which can be replaced with gas-efficient bytecodes to save money. In particular, we identify 7 gas-costly patterns belonging to 2 categories. Moreover, we propose and develop GASPER that leverages symbolic execution to automatically discover 3 representative gas-costly patterns in bytecode. By applying GASPER to all deployed smart contracts until Nov. 5th, 2016, we find that 93.5%, 90.1% and 80% smart contracts suffer from these 3 patterns, respectively. In future work, we will extend this research from the following aspects: (1) identifying more gas-costly patterns and the corresponding gas-efficient patterns; (2) extending GASPER to cover all these patterns; (3) improving compilers to produce gas-efficient bytecode.

ACKNOWLEDGEMENT

This work is supported in part by the Hong Kong GRF (PolyU 152279/16E), the HKPolyU Research Grants (G-YBJX), Shenzhen City Science and Technology R&D Fund (No. JCYJ20150630115257892), the National Natural Science Foundation of China (No.61402080, No.61572115, No.61502086, No.61572109), and China Postdoctoral Science Foundation funded project (No.2014M562307).

REFERENCES

- [1] J. Redman, (Jun., 2016) Bitcoin price rally rages on, market cap passes \$10bn usd. [Online]. Available: <https://news.bitcoin.com/bitcoin-price-market-cap-10-billion/>
- [2] (Nov., 2016) Etherscan, transactions. [Online]. Available: <https://etherscan.io/txs>
- [3] (Nov., 2016) Etherscan, gasprice history. [Online]. Available: <https://etherscan.io/charts/gasprice>
- [4] (Nov., 2016) Buy ethereum: Eth/btc, eth/usd and eth/eur. [Online]. Available: <https://cex.io/buy-ethereum>
- [5] G. Wood, “Ethereum: A secure decentralised generalised transaction ledger,” 2014.
- [6] L. De Moura and N. Bjørner, “Z3: An efficient smt solver,” in *Proc. TACAS*, 2008.
- [7] L. Luu, D. H. Chu, H. Olickel, P. Saxena, and A. Hobor, “Making smart contracts smarter,” in *Proc. CCS*, 2016.
- [8] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, N. Gonthier, G. and Kobeissi, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Beguelin, “Formal verification of smart contracts: Short paper,” in *Workshop. PLAS*, 2016.
- [9] J.-C. Filliâtre and A. Paskevich, “Why3-where programs meet provers,” in *Proc. ESOP*, 2013.
- [10] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou, “Hawk: The blockchain model of cryptography and privacy-preserving smart contracts,” in *Proc. S&P*, 2016.
- [11] A. Juels, A. Kosba, and E. Shi, “The ring of gyges: Investigating the future of criminal smart contracts,” in *Proc. CCS*, 2016.
- [12] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi, “Town crier: An authenticated data feed for smart contracts,” in *Proc. CCS*, 2016.
- [13] N. Atzei, M. Bartoletti, and T. Cimoli, “A survey of attacks on ethereum smart contracts,” 2016.