Gbadebo Ayoade, Erick Bauman, Latifur Khan, and Kevin W. Hamlen. "Smart Contract Defense Through Bytecode Rewriting." In *Proc. IEEE Int. Conf. Recent Advances on Blockchain and its Applications (ICBC)*, pp. 384-389, July 2019

Smart Contract Defense Through Bytecode Rewriting

Gbadebo Ayoade, Erick Bauman, Latifur Khan, and Kevin W. Hamlen
Department of Computer Science
The University of Texas at Dallas
{*gbadebo.ayoade, erick.bauman, lkhan, hamlen*}@*utdallas.edu*

***Abstract*—An Ethereum bytecode rewriting and validation architecture is proposed and evaluated for securing smart contracts in decentralized cryptocurrency systems without access to contract source code. This addresses a wave of smart contract vulnerabilities that have been exploited by cybercriminals in recent years to steal millions of dollars from victims. Such attacks have motivated various best practices proposals for helping developers write safer contracts; but as the number of programming languages used to develop smart contracts increases, implementing these best practices can be cumbersome and hard to enforce across the development tool chain. Automated hardening at the bytecode level bypasses this source-level heterogeneity to enforce safety and code integrity properties of contracts independently of the sources whence they were derived. In addition, a binary code verification tool implemented atop the Coq interactive theorem prover establishes input-output equivalence between the original code and the modified code. Evaluation demonstrates that the system can enforce policies that protect against integer overflow and underflow vulnerabilities in real Ethereum contract bytecode, and overhead is measured in terms of instruction counts.**

***Keywords*-blockchain; Ethereum; in-lined reference monitors; formal methods**

I. INTRODUCTION

Recent increases in the adoption rate of smart contract applications have spurred initial coin offerings (ICOs)[1] and decentralized autonomous organizations (DAOs) to leverage multiple applications to raise money for disparate start-ups. This surge in investment has motivated a corresponding surge in smart contract attacks and vulnerability discoveries. For example, cybercriminals have leveraged re-entrancy attacks [1], [2] and parity multisig wallet attacks [3] to steal more than 60 million dollars in cryptocurrency.

As a result, various languages have been developed or modified to compile smart contracts as Ethereum bytecode. These include Solidity[2] (which resembles JavaScript), Haskell,[3] and Vyper.[4] Solidity is presently the most popular of these languages. However, developers are often reluctant to learn new languages, and gaining the proficiency to develop correct and secure code in a new language can be demanding.

These obstacles are exacerbated by the increasing complexity and subtlety of vulnerabilities leveraged by attackers to exploit and steal cryptocurrencies from blockchain networks. Various researchers have proposed automated tools for finding bugs in smart contracts before deployment to the blockchain network. Most of these tools rely on the source code to carry out their analysis [4], [5], though a few (e.g., teEther [6]) perform bug-search at the bytecode level.

Rather than searching for bugs, our work leverages automated bytecode rewriting to allow developers to create smart contracts in any language, yet automatically enforce security policies at the bytecode level without relying on developer expertise to secure the application. Our framework ensures that vulnerable bytecode is properly protected without access to source code. By providing a framework that uses a source-agnostic approach, we can enforce security policy rules across different development tool chains.

Source-free, binary transformations are widely recognized as more difficult to implement than source-level analyses and transformations. Lack of contextual variable meanings [7], irregular instruction alignment of certain architectures (e.g., CISC native codes) [8], and recovery of code control-flow graphs and function entry points [9], are all perennial challenges documented in the literature. However, Ethereum bytecode has many syntactic properties that aid feasibility of binary rewriting of smart contracts relative to other binary languages, including strict instruction alignment and whitelisting of all indirect control-flow targets with `JUMPDEST` opcodes [10].

Bytecode rewriting of Ethereum contracts can therefore be achieved in four major steps: (1) Disassemble the bytecode to semantically equivalent assembly code. (2) Instrument the disassembled bytecode with new security guard code that enforces the desired policy. (3) Identify all jump locations and rewrite their destinations to match the code motions induced by the instrumentation step. (4) Verify that the modified code is *transparent* [11] with respect to the original code (i.e., it implements the same input-output relation whenever the security policy is not violated).

In this work, we build a framework that can rewrite Ethereum bytecode and update all jump instructions to reflect the new offset of their targets based on the modified code. Our work differs from previous systems by creating a framework that can modify the Ethereum bytecode without the need of high level language source code (cf., [4], [12]). In short, our contributions include the following.

---

[1] https://www.icohotlist.com
[2] https://github.com/ethereum/solidity
[3] https://github.com/takenobu-hs/haskell-ethereum-assembly
[4] https://github.com/ethereum/vyper

- We propose and implement a framework to rewrite Ethereum bytecode without access to source code.
- Our framework detects vulnerable bytecode instructions and inserts guard code to mitigate attacker exploits.
- We implement Ethereum virtual machine code verification in the Coq theorem prover to machine-prove semantic transparency.
- We evaluate the system on real world smart contracts and measure the system overhead.

The rest of the paper is organized as follows. Section II provides background on smart contract and Ethereum bytecode vulnerabilities. Section III discusses challenges and solutions encountered in designing a bytecode rewriter framework. Section IV provides the architecture of our system and Section V describes our implementation. Section VI contains our evaluation, followed by discussion of related work in Section VII. Finally, Section VIII examines limitations and proposes future directions, and Section IX concludes.

II. BACKGROUND

A. Ethereum Virtual Machine

Smart contracts are autonomous computations executed by decentralized entities on a blockchain. A popular smart contract framework implementation is the Ethereum virtual machine (EVM). The EVM is a stack based computer that executes a sequence of bytecode instructions. Its state consists of a stack of 32-byte values, a memory, and a key-value store for persistent storage. EVM bytecode consists of more than 100 opcodes, such as `ADD`, `SUB`, `PUSH`, and `JUMP`. Each opcode has an associated fee called *gas* that must be paid to execute the instruction. Two token standards called ERC20 and ERC721 implement custom cryptocurrencies and custom non-fungible assets.

B. Common Ethereum Smart Contract Vulnerabilities

The increased use of smart contracts has resulted in the discovery of numerous contract vulnerabilities, including arithmetic over/underflow, smart contract owner hijacking, and re-entrancy attack vulnerabilities. We here focus on arithmetic vulnerability detection and mitigation. Integer overflows and underflows can occur during EVM code execution, leading to loss of tokens or money. The stack consists of up to 1024 32-byte words, each of which can hold a maximum value of 2^{256}. By adding a number to the max value, the new value rolls over to zero. Subtracting from a zero value dually rolls the result over to the maximum value[1] [13] because EVM uses unsigned int256 types [2]. Integer underflow vulnerabilities can allow an attacker to roll over his initial balance to the maximum value, thereby gaining access to a large token balance that he does not own. This work focuses on mitigating these vulnerabilities by rewriting the smart contract bytecode.

III. CHALLENGES

A. EVM Control-flows and Jump Retargeting

Since the EVM is a stack-based machine, EVM bytecode consists of a sequence of one-byte instructions (except for `PUSH` instructions, which contain immediate values). To control the flow of the program, the address of a jump destination is first pushed to the stack as an input to the jump instruction, which is then executed. All jump destination addresses are marked with the `JUMPDEST` instruction. This is to ensure that programs can only jump to specific unique addresses marked in the bytecode. This mechanism is enforced by the EVM. To enforce this policy, the EVM parses the program bytecode and memorizes all the `JUMPDEST` targets. Every jump target is checked for validity before executing each `JUMP` instruction.

Unlike most native code architectures, where the machine code contains direct jump instructions, all jumps in EVM bytecode use the address at the top of the stack to identify the jump target. Code transformations that move instructions must therefore modify all jumps whose targets might have moved. We address this challenge in Section IV-A.

B. Minimizing Overhead in Modified Bytecode

Protecting vulnerable code segments in the bytecode requires adding more instructions to the original bytecode. Inserting full guard code where vulnerable code exists results in a larger bytecode size, which affects the deployment cost on the Ethereum blockchain. For example, a smart contract with 100 bytes of code costs 2000 gas to deploy, while a smart contract with 50 bytes of code costs 1000 gas, resulting in a savings of 50%. According to the Ethereum yellow paper [10], every byte deployed on the blockchain costs 200 gas. As a result of this, we need an efficient technique to optimize the rewriting. We address this challenge in Section IV-C.

C. Verifying Bytecode Correctness and Transparency

Bytecode rewriting is a potentially complex operation. To obtain high assurance, a machine-checked verification system allows us to verify that the modified bytecode program maintains the policy-compliant behaviors and correctness properties of the original code. To address this need, we build a verification tool that simulates the Ethereum stack VM and prove the transparency of the original and the modified bytecode. We address this challenge in Section IV-D.

IV. ARCHITECTURE

As shown in Figure 1, our system accepts policy rules and EVM bytecode as input. The framework consists of the Bytecode rewriter and the EVM code verifier. The bytecode rewriter consists of a disassembler, the rewriter, and an assembler. The bytecode rewriter output is fed to the EVM code verifier together with the original bytecode to ensure the rewritten program is equivalent. If the verifier succeeds, we output the hardened bytecode. If the verifier fails, we retry the

---

[1] https://github.com/CoinCulture/evm-tools/blob/master/analysis/guide.md

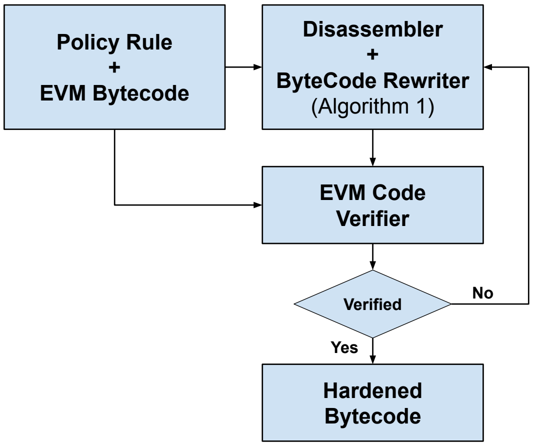

Figure 1: System architecture

Algorithm 1: EVM bytecode hardening

```
Data: EVMbytecode, EVMGuardCode, VulnerableOpcode
Result: HardenedEVMBytecode
1  while $Inst \in Instructions$ do
2      Opcode := GetOpcode(Inst);
3      Operand := GetOperand(Inst);
4      if $VulnerableOpcode = Opcode$ then
5          InsertCode(EVMGuardCode);
6  while $Inst \in Instructions$ do
7      Opcode := GetOpcode(Inst);
8      Operand := GetOperand(Inst);
9      if $Opcode = JUMP$ then
10         pushInst := GetPreviousPushInst();
11         oldTarget := getOperand(pushInst);
12         while $Inst_2 \in Instructions$ do
13             if $oldTarget = oldlabel(Inst_2) \wedge GetOpcode(Inst_2) = JUMPDEST$ then
14                 rewritePushInstruction(Inst, getnewlabel($Inst_2$))
```

rewriting step with a bounded time. In our system, we propose an in-lined bytecode insertion algorithm shown in Algorithm 1 and a function call technique shown in Algorithm 2.

A. In-lined Bytecode Rewriter

In order to rewrite the EVM bytecode as shown in Algorithm 1, we first disassemble the bytecode to opcode instructions and extract the opcode and the operand of each instruction (lines 2–3). If the opcode is vulnerable, we insert the guard code before the vulnerable opcodes (line 5).

The resulting assembly code is misaligned due to the inserted code and `JUMP` instructions. To realign the code, we scan the code again for `JUMP` or `JUMPI` instructions (line 9). In most cases the instruction that precedes the `JUMP` opcode is a `PUSH` instruction, whereupon we extract the argument that specifies the jump target (line 11). We next scan the instructions to find a match between the extracted old jump target and the new jump target location label (line 12). After a match is found, we extract the offset of the new `JUMPDEST` instruction and rewrite the `PUSH` instruction's argument to the new jump target location (line 14).

```
DUP1 // duplicate second subtraction argument
DUP3 // duplicate first subtraction argument
GT // test for underflow
NOT
PUSH [tag] n
JUMPI
REVERT // underflow detected
tag n
JUMPDEST
SUB // safely perform subtraction
```

Listing 1: Underflow protection bytecode

B. Addressing the Policy Rule Generation Challenge

We must generate and convert a protection policy rule to bytecode, which can then be used as guard code to protect the vulnerable code. Since we perform our rewriting at the bytecode level, we can generate the guard bytecode once and apply it to any other bytecode compiled from any other language. In our case, we write our protection policy in Solidity and extract the compiled bytecode for the application.

For example, Listing 1 shows the code generated for underflow code protection. The code checks whether the subtrahend exceeds the minuend before performing the subtraction to avoid a negative result, which is not supported by the EVM as discussed in Section II-B [2], [13]. Program execution is aborted if an impending underflow is detected.

C. Optimized Guard Code Rewrite

Algorithm 2 addresses the challenge of optimizing the bytecode rewriting algorithm to minimize bytecode size and instruction count, as mentioned in Section III-B. In order to minimize the size of the binary file generated by inline guard code insertion, we utilized a function call-like system in the EVM bytecode. EVM does not support first-class function calls at the bytecode level.

To achieve our optimization, we inserted code that allows the program to remember how to return to the calling function after executing the guard code. In order to achieve this, function call code is first inserted before all vulnerable instruction code (line 5). Guard code is next appended to the current bytecode (line 6).

The function call code's `PUSH` argument is initialized with a placeholder location value that is later updated, the instruction is labeled as the current location, and the consecutive instruction is labeled as the function call instruction. To update the place holder location in the function call, we first scan the new code for the location of the appended guard code. Second, we scan the code for the labels; if the instruction label is the current location (line 9), we update the `PUSH` argument to the current location value to save the return

Algorithm 2: EVM bytecode optimized rewriter

```
   Data: EVMbytecode, EVMGuardCode, Vulnerable_Opcode
   Result: HardenedEVMBytecode
1  while $Inst \in Instructions$ do
2      Opcode := GetOpcode(Inst);
3      Operand := GetOperand(Inst);
4      if $VulnerableOpcode = Opcode$ then
5          InsertFunctionCallCode();
6  AppendCode(EVMGuardCode);
7  while $Inst \in Instructions$ do
8      instructionLabel := getInstructionLabel();
9      if $instructionLabel = saveCurrentInstAddress$ then
10         UpdatePushInstrArg(Inst, currentLocation);
11     if $currInstructionLabel = functioncall$ then
12         UpdatePushInstrArg(Inst, appendedCodeLocation);
13 Reuse steps 6–14 of Algorithm 1 to rewrite jump targets
```

```
000000 PUSH ⟨current address⟩
000002 PUSH ⟨address of appended guard code⟩
000004 JUMP
```

Listing 2: Function call code in EVM bytecode

address on the stack. Third, we scan the instructions for the function call `PUSH` instruction and we update the argument to the location of the appended guard code (lines 12). Finally, we rewrite the jump target locations using the steps from Algorithm 1.

Listing 2 shows the code flow of the function call code routine. To call the appended guard code, the current address of the program instruction is first pushed to the stack. Second, the address of the guard code function is pushed to the stack and the jump instruction is executed to the jump to the guard code. After execution, the guard uses the saved location to return to the calling code position.

D. EVM Code Verification

Here we address the challenge of bytecode verification as introduced in Section III-C. In order to verify the properties of the modified bytecode are still correct, we need a system that allows us to specify theorems about program behaviors and prove their correctness. By leveraging the Coq interactive proof assistant, we implemented an EVM stack in Coq.

Figure 2 shows a simplified and abbreviated definition of our EVM semantics for Coq. The semantics are formalized as a small-step machine in which bad states (e.g., stack underflows, invalid jumps, etc.) are intentionally left undefined. This makes unprovable any theorems that depend upon the EVM's behavior upon encountering such states. As a result, proved theorems guarantee that bad states are avoided.

$$\begin{array}{lr}
\rho : \mathbb{N} \rightharpoonup \iota & \text{(program)} \\
\iota ::= \texttt{PUSH}\ n \mid \texttt{POP} \mid \texttt{SUB} \mid \texttt{JUMP} \mid \texttt{STOP} \mid \cdots & \text{(instruction)} \\
\mu ::= \langle \sigma, \mathbf{m} \rangle \mid \langle pc, \sigma, \mathbf{m} \rangle & \text{(machine state)} \\
\sigma ::= \cdot \mid n :: \sigma & \text{(stack)} \\
\mathbf{m} : \mathbb{N} \rightharpoonup \mathbb{N} & \text{(memory)}
\end{array}$$

$$\frac{\rho(pc) = \texttt{PUSH}\ n}{\rho \vdash \langle pc, \sigma, \mathbf{m} \rangle \to_1 \langle pc+1, n :: \sigma, \mathbf{m} \rangle}\text{(PUSH)}$$

$$\frac{\rho(pc) = \texttt{POP}}{\rho \vdash \langle pc, n :: \sigma, \mathbf{m} \rangle \to_1 \langle pc+1, \sigma, \mathbf{m} \rangle}\text{(POP)}$$

$$\frac{\rho(pc) = \texttt{SUB} \qquad n_1 \geq n_2}{\rho \vdash \langle pc, n_1 :: n_2 :: \sigma, \mathbf{m} \rangle \to_1 \langle pc+1, (n_1 - n_2) :: \sigma, \mathbf{m} \rangle}\text{(SUB)}$$

$$\frac{\rho(pc) = \texttt{JUMP} \qquad \rho(n) = \texttt{JUMPDEST}}{\rho \vdash \langle pc, n :: \sigma, \mathbf{m} \rangle \to_1 \langle n, \sigma, \mathbf{m} \rangle}\text{(JUMP)}$$

$$\frac{\rho(pc) = \texttt{STOP}}{\rho \vdash \langle pc, \sigma, \mathbf{m} \rangle \to_1 \langle \sigma, \mathbf{m} \rangle}\text{(STOP)}$$

Figure 2: EVM semantics (abbreviated and simplified)

A program ρ is formalized as a partial mapping from offsets to instructions ι. The program's current state includes the current instruction offset (program counter pc), the stack contents σ, and the memory contents $\mathbf{m}$. Each semantic rule executes one instruction by reading the opcode located at the current program counter offset and manipulating the stack and/or memory accordingly. Fall-through instructions increment the program counter, whereas jumps assign it a target offset. Programs that halt normally enter final state $\langle \sigma, \mathbf{m} \rangle$.

E. Proving Transparency

Proving transparency of our bytecode rewriter entails proving that all policy-adherent behaviors of the program are preserved after rewriting. For a given rewriter $R : \rho \to \rho$, the transparency theorem can be formalized as follows.

Theorem 1: For all programs ρ, if $\rho \vdash \langle 0, \cdot, \mathbf{m} \rangle \to^* \langle \sigma', \mathbf{m}' \rangle$ is derivable, then $R(\rho) \vdash \langle 0, \cdot, \mathbf{m} \rangle \to^* \langle \sigma', \mathbf{m}' \rangle$ is derivable, where $\to^*$ is the reflexive, transitive closure of small-step relation $\to_1$.

Proof: The theorem is proved by first generalizing the theorem statement's initial program counter for the original program (0) to an arbitrary offset pc, and rewriting the rewritten program's initial program counter 0 to $r(pc)$, where $r : \mathbb{N} \to \mathbb{N}$ is the mapping from old offsets to new (relocated) offsets implemented by R. Initial stack $\cdot$ is likewise generalized to an arbitrary stack σ. This generalization of the theorem facilitates a natural number induction over the number of steps n in transitive relation $\to^*$. By case distinction, each small-step semantic rule in Figure 2 yields a modified state that satisfies the theorem by inductive hypothesis. The rule for instruction `STOP` satisfies the base case of the induction, completing the proof. ■

V. IMPLEMENTATION

We implemented our bytecode rewriter in Python. We utilized the Ethereum dataset of all smart contract bytecode stored on the Google big-query platform. We extracted a total of 155,175 unique smart contracts from a total of

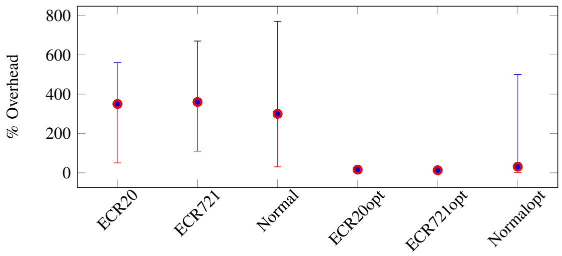

Figure 3: MIN, AVG and Max instruction count overhead for integer overflow protection rewrite

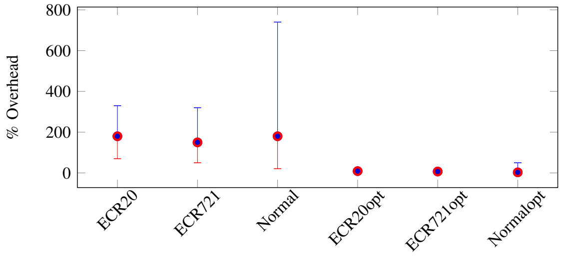

Figure 4: MIN, AVG and Max instruction count overhead for integer underflow protection rewrite

2,195,890 smart contracts deployed on the Ethereum network. Of the 155,175 smart contracts, we extracted 64,033 ECR20 Ethereum smart contracts and 1,515 ECR721 Ethereum smart contracts. For each of the smart contract types, we instrumented 1,000 smart contracts with code protection for integer overflow and underflow for both addition and subtraction instructions. We executed our bytecode rewriter on an Intel Core i5 with 8GB of memory.

We implemented our EVM verification by extending a stack computer developed in Coq [14]. As discussed in Section IV-D, we implemented the following instructions in Coq: `PUSH`, `ADD`, `SUB`, `MULT`, `POP`, `DIV`, `LT`, `GT`, `DUP1`, `DUP2`, `SWAP1`, `SWAP2`, `EXP`, `ISZERO`, and `STOP`. This subset encompasses all instructions needed for our guard code implementations. Since all other EVM opcodes are preserved by our rewriter, their semantics are not needed in the proof of Theorem 1.

VI. EVALUATION

In this section, we discuss the overhead in terms of instruction counts, as shown in Figures 3 and 4. The x-axis lists the different types of Ethereum smart contract interfaces as ECR20, ECR720, and normal smart contracts. The y-axis records the overhead as the increase in the instruction count of the modified code relative to the original code, giving the minimum, average, and maximum overhead for each. The non-optimized result represents the in-lined rewriting algorithm, and the optimized result represents the function call method.

Protection	**ECR20**	**ECR721**	**Norm**
Overflow	350%	360%	300%
Underflow	180%	150%	180%
Protection	**ECR20opt**	**ECR721opt**	**Normopt**
Overflow	16%	13%	31%
Underflow	9%	7%	3%

Table I: Average instruction count overheads

For Figure 3, the minimum instruction count percentage overhead for normal smart contracts is 30% for the non-optimized rewriter and 2% for the optimized rewriter. The average percentage overhead is 300% for non-optimized versus 31% for the optimized rewriter for normal smart contracts. Figure 4 shows similar results. Table I contains the average overheads for each type of contract.

To evaluate the overhead based on gas usage, we used the EVM simulator program developed by the Ethereum foundation to run a normal smart contract that is vulnerable to integer overflow and integer underflow, and we compared the gas usage to the smart contract protected by our rewriting framework. The execution overhead for the protected program from integer overflow and underflow is 300% for both. This result is due to similar instructions used to check if the parameters for addition or subtraction will not roll over to a zero or a maximum number as discussed in Section II-B.

VII. RELATED WORK

Smart Contract Code Defense. Various researchers have explored finding vulnerabilities in smart contracts. Oyente [4] uses symbolic execution to run smart contracts and find predefined bugs such as re-entrancy attacks and insufficient balance. Osiris [15] extends Oyente to discover arithmetic vulnerabilities by using taint propagation techniques. Machine learning [16] based methods to classify code to detect vulnerable code have been explored. TeEther [6] allows automatic generation of exploits for smart contracts. Zeus [5] leverages LLVM to generate LLVM IR code from an abstract syntax tree generated from solidity source code. The system needs access to the original source code to mitigate attacks.

Formal Verification. Machine-verifying the correctness of security-sensitive programs for high assurance is becoming more important with the increase in security breaches. One main work in the area of program verification is compiler certification. Coq has been used to develop the first C compiler with an end-to-end, machine-checked proof of semantic transparency [17]. In order to verify the safety properties of smart contracts in Ethereum, an Ethereum smart contract verification system has been implemented in Isabelle/HOL [18]. Our work differs from these works by developing a framework that provably mitigates smart contract vulnerabilities by inserting guard code in the raw bytecode.

VIII. DISCUSSION AND FUTURE WORK

In this work, we identified arithmetic vulnerabilities by searching for the occurrence of `ADD` and `SUB` instructions. Other instructions, such as the `SIGNEXTEND` opcode, can also be used to determine the bitwidth [15] and type inference of the result address. This instruction can help to determine whether an arithmetic overflow will occur. In addition, we identified jump target locations where the address of the `JUMP` instruction is pushed to the stack before the jump is executed.

For our formal verification, we focused mainly on verifying the operations of the Ethereum stack where most of the arithmetic operations occur. In future work, we will focus on adding the verification of memory access operations that can be useful in protecting against other Ethereum smart contract vulnerabilities.

For future work, we will identify other common jump operation patterns that involve function call patterns. In addition, we will use machine learning methods to detect vulnerabilities in smart contract bytecode.

IX. CONCLUSION

This work explored bytecode rewriting as a mechanism for defending against smart contract vulnerabilities. Hardened EVM bytecode exhibited an average overhead of between 3% and 31% for both integer overflow and integer underflow guard code rewriting using our optimized bytecode rewriter. In addition, we implemented a code verification system within the Coq interactive theorem prover to machine-verify the transparency of the modified bytecode.

X. ACKNOWLEDGMENTS

This research was supported in part by NSF awards DMS-1737978, DGE 17236021, OAC-1828467, and 1513704; ARO award W911-NF-18-1-0249; an IBM faculty award (Research); an award from NSA; ONR award N00014-17-1-2995; DARPA award FA8750-19-C-0006; and a gift from the Eugene McDermott family.

REFERENCES

[1] N. Atzei, M. Bartoletti, and T. Cimoli, “A survey of attacks on Ethereum smart contracts SoK,” in *Proc. 6th Int. Conf. Principles of Security and Trust – Volume 10204*, 2017, pp. 164–186.

[2] G. Konstantopoulos, “How to secure your smart contracts: 6 solidity vulnerabilities and how to avoid them (part 1),” *Loom Network J.*, January 2018, https://bit.ly/2nNLuOr.

[3] L. Breidenbach, P. Daian, A. Juels, and E. G. Sirer, “An in-depth look at the Parity Multisig bug,” Hacking, Distributed, July 2017, http://hackingdistributed.com/2017/07/22/deep-dive-parity-bug.

[4] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, “Making smart contracts smarter,” in *Proc. 23rd ACM Conf. Computer and Communications Security (CCS)*, 2016, pp. 254–269.

[5] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, “Zeus: Analyzing safety of smart contracts,” in *Proc. 25th Annual Network & Distributed System Security Sym. (NDSS)*, 2018.

[6] J. Krupp and C. Rossow, “teEther: Gnawing at Ethereum to automatically exploit smart contracts,” in *Proc. 27th USENIX Security Sym.*, 2018, pp. 1317–1333.

[7] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Krügel, and G. Vigna, “Ramblr: Making reassembly great again,” in *Proc. 24th Annual Network & Distributed System Security Sym. (NDSS)*, 2017.

[8] E. Bauman, Z. Lin, and K. W. Hamlen, “Superset disassembly: Statically rewriting x86 binaries without heuristics,” in *Proc. 25th Annual Network & Distributed System Security Sym. (NDSS)*, 2018.

[9] R. Wartell, Y. Zhou, K. W. Hamlen, and M. Kantarcioglu, “Shingled graph disassembly: Finding the undecidable path,” in *Proc. 18th Pacific-Asia Conf. Knowledge Discovery and Data Mining (PAKDD)*, 2014, pp. 273–285.

[10] G. Wood, “Ethereum: A secure decentralised generalised transaction ledger,” Ethereum & Parity, Tech. Rep. 3e36772, 2019.

[11] M. Sridhar, R. Wartell, and K. W. Hamlen, “Hippocratic binary instrumentation: First do no harm,” *Science of Computer Programming (SCP), Special Issue on Invariant Generation*, vol. 93, no. B, pp. 110–124, November 2014.

[12] J. Lind, I. Eyal, P. Pietzuch, and E. G. Sirer, “Teechan: Payment channels using trusted execution environments,” arXiv Preprint 1612.07766, 2016.

[13] ConsenSys, “Ethereum smart contract best practices: Known attacks,” https://consensys.github.io/smart-contract-best-practices/known_attacks, 2019.

[14] F. Z. Nardelli, “Modelling and verifying algorithms in Coq: An introduction,” https://www.di.ens.fr/~zappa/teaching/coq/ecole11/summer/exercices/solutions/compiler-sol.v, 2011.

[15] C. F. Torres, J. Schütte, and R. State, “Osiris: Hunting for integer bugs in Ethereum smart contracts,” in *Proc. 34th Annual Computer Security Applications Conf. (ACSAC)*, 2018, pp. 664–676.

[16] M. M. Masud, L. Khan, and B. Thuraisingham, “A scalable multi-level feature extraction technique to detect malicious executables,” *Information Systems Frontiers*, vol. 10, no. 1, pp. 33–45, 2008.

[17] X. Leroy, “Formal verification of a realistic compiler,” *Communications ACM (CACM)*, vol. 52, no. 7, pp. 107–115, 2009.

[18] Y. Hirai, “Defining the Ethereum virtual machine for interactive theorem provers,” in *Proc. Int. Conf. Financial Cryptography and Data Security (FC)*, 2017, pp. 520–535.