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) and decentralized autonomous organizations (DAOs) to leverage multiple applications to raise money for disparate start-ups. This surge in investment has motivated a corresponding release of more than 60 million 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.

Contributions include the following.

1. Various obstacles are exacerbated by the increasing complexity and subtlety of vulnerabilities leveraged by attackers to exploit and steal cryptocurrencies from blockchain networks.

2. 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.

3. 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.

4. 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].

5. 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).

6. Bytecode rewriting of 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.

I. INTRODUCTION

Recent increases in the adoption rate of smart contract applications have spurred initial coin offerings (ICOs) and decentralized autonomous organizations (DAOs) to leverage multiple applications to raise money for disparate start-ups. This surge in investment has motivated a corresponding release of more than 60 million 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.

Contributions include the following.

1. Various obstacles are exacerbated by the increasing complexity and subtlety of vulnerabilities leveraged by attackers to exploit and steal cryptocurrencies from blockchain networks.

2. 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.

3. 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.

4. 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].

5. 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).

6. Bytecode rewriting of 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.
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
Algorithm 1: EVM bytecode hardening

Data: EVM bytecode, EVM guard code, Vulnerable Opcode
Result: Hardened EVM bytecode

```plaintext
1 while Inst ∈ Instructions do
2    Opcode := GetOpcode(Inst);
3    Operand := GetOperand(Inst);
4    if Vulnerable Opcode = Opcode then
5       InsertCode(EVMSecurityCode);
6    end
7    while Inst ∈ Instructions do
8       Opcode := GetOpcode(Inst);
9       Operand := GetOperand(Inst);
10      if Opcode = JUMP then
11         pushInst := GetPreviousPushInst();
12         oldTarget := getOperand(pushInst);
13         while Inst ∈ Instructions do
14            if oldTarget = oldlabel(Inst) ∧ GetOpcode(Inst) = JUMPI
15               rewritePushInstruction(Inst, getnewlabel(Inst));
```

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 placeholder location value, 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 match is found, we extract the offset of the new JUMPI instruction and rewrite the PUSH instruction’s argument to the new jump target location (line 14).

Listing 1: Underflow protection bytecode

1  DUP1  // duplicate second subtraction argument
2  DUP3  // duplicate first subtraction argument
3  GT    // test for underflow
4  NOT   
5  PUSH [tag] n
6  JUMP  
7  REVERT // underflow detected
8  tag n  
9  JUMPDEST 
10  SUB   // safely perform subtraction

Figure 1: System architecture
Algorithm 2: EVM bytecode optimized rewriter

Data: EVMBytecode, EVMGuardCode, VulnerableOpcode
Result: HardenedEVMBytecode

1 while Inst ∈ Instructions do
2   Opcode := GetOpcode(Inst);
3   Operand := GetOperand(Inst);
4   if VulnerableOpcode = Opcode then
5     InsertFunctionCallCode();
6   end
7   AppendCode(EVMGuardCode);
8   if instructionLabel = getInstructionLabel() then
9     UpdatePushInstrArg(Inst, currentLocation);
10    UpdatePushInstrArg(Inst, appendedCodeLocation);
11   end
12   if currInstructionLabel = functioncall then
13     AppendCode(EVMGuardCode);
14 end
15 Reuse steps 6–14 of Algorithm 1 to rewrite jump targets

address on the stack. Third, we scan the instructions for the function call PUSH instruction and 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.

A program $\rho$ is formalized as a partial mapping from offsets to instructions $i$. The program’s current state includes the current instruction offset (program counter $pc$), the stack contents $\sigma$, and the memory contents $m$. 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 $(\sigma, m)$.

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 $R: \rho \rightarrow \rho$, the transparency theorem can be formalized as follows.

Theorem 1: For all programs $\rho$, if $\rho \vdash (0, \cdot, m) \rightarrow^* (\sigma, m')$ is derivable, then $R(\rho) \vdash (0, \cdot, m) \rightarrow^* (\sigma', m'')$ is derivable, where $\rightarrow^*$ is the reflexive, transitive closure of small-step relation $\rightarrow_1$.

Proof: The theorem is proved by first generalizing the theorem statement’s initial program counter for the original program $(0) \rightarrow$ to an arbitrary offset $pc$, and rewriting the rewritten program’s initial program counter $0 \rightarrow r(pc)$, where $r: N \rightarrow N$ is the mapping from old offsets to new (relocated) offsets implemented by $R$. Initial stack $\cdot$ is likewise generalized to an arbitrary stack $\sigma$. This generalization of the theorem facilitates a natural number induction over the number of steps $n$ in transitive relation $\rightarrow^*$. 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
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.

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


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 1723602, and 1513704; DARPA award FA8750-19-C-0006; and a gift from the Eugene McDermott family.

REFERENCES


