EXGEN: Cross-platform, Automated Exploit Generation for Smart Contract Vulnerabilities

Ling Jin, Yinzhi Cao, Yan Chen, Fellow, IEEE, Di Zhang, and Simone Campanoni

Abstract—Smart contracts, just like other computer programs, are prone to a variety of vulnerabilities, which lead to severe consequences including massive token and coin losses. Prior works have explored automated exploit generation for vulnerable Ethereum contracts. However, the scopes of prior works are limited in both vulnerability types and contract platforms. In this paper, we propose a cross-platform framework, called EXGEN, to generate multiple transactions as exploits to given vulnerable smart contracts. EXGEN first translates either Ethereum or EOS contracts to an intermediate representation (IR). Then, EXGEN generates symbolic attack contracts with transactions in a partial order and then symbolically executes the attack contracts together with the target to find and solve all the constraints. Lastly, EXGEN concretizes all the symbols, generates attack contracts with multiple transactions, and verifies the generated contracts' exploitability on a private chain with values crawled from the public chain. We implemented a prototype of EXGEN and evaluated it on Ethereum and EOS benchmarks. EXGEN successfully exploits 1,258/1,399 (89.9%) Ethereum and 126/130 (96.9%) EOS vulnerabilities. EXGEN is also able to exploit zero-day vulnerabilities on EOS.

Index Terms—Blockchain, Smart Contract, Automated Exploit Generation, Symbolic Execution

1 INTRODUCTION

Smart contracts are computer programs running on blockchains, such as Solidity code on Ethererum and C/C++ code on EOS. Just like other computer programs, smart contracts are also prone to a variety of vulnerabilities, such as reentrancy [1] and integer overflows [2]. More importantly, because such vulnerabilities often have cryptocurrencies involved, the consequence of a compromise is severe. For example, a security incident [3] related to integer overflow in July 2018 causes a loss of 60,686 EOS coins worthing about $515,831. A popular yet powerful technique in detecting smart contract vulnerabilities is static analysis and in the past researchers have extensively proposed static analysis tools [4]–[10] to find hundreds of smart contract vulnerabilities. One emerging problem of static vulnerability detection is that the reported vulnerabilities may not be exploitable because the found path could be unsatisfiable.

Generally speaking, there are two types of approaches, dynamic and static, to generate exploits for vulnerabilities. On one hand, dynamic analysis can be used to either fuzz smart contracts or analyze real-world smart contract transactions. For example, Zhou et al. [11] analyzes real-world transactions using graph-based signatures for vulnerabilities. EasyFlow [12] propagates taint information during dynamic execution of Ethereum contracts and also generates transactions with large or small integers as fuzzing inputs to trigger integer overflows. Similarly, Zhang et al. [13] and Jiang et al. [14] also fuzz Ethereum smart contracts to find and trigger vulnerabilities. However, dynamic analysis, especially fuzzing, is known to have code coverage issues, i.e., it may not trigger the vulnerable code if certain constraints are not satisfied.

On the other hand, researchers have also proposed static automated exploit generation on smart contracts: The state-of-the-art work is called teEther [15], which relies on symbolic execution to generate a transaction triggering the vulnerable condition on Ethereum. However, its scopes are limited in both vulnerability types and contract platforms. That is, teEther is only applicable to suicidal and call injection on Ethereum, but not other popular vulnerabilities, such as integer overflow and reentrancy, let alone those on EOS. The fundamental reasons are two-fold. First, teEther can only generate transactions that have straight dependencies in a line, which is called a path in their paper (e.g., Transaction A depending on B and then C), while many vulnerabilities require non-straight dependencies (e.g., Transaction C depending on both transactions A and B). Second, teEther does not support dataflow constraints, which are required for many vulnerability types. Take integer overflow for example. A successful exploitation requires not only the execution of vulnerable statement but the value, e.g., the addition of two inputs, overflows the vulnerable variable.

In this paper, we propose the first cross-platform, static, automated exploit generation framework, called EXGEN, for smart contract vulnerabilities. The key insight of EXGEN is that data dependencies among transactions are transitive, which can be modeled as a partially-ordered set, called Partially-ordered Transactional Set (PTS) in the paper. PTS is non-flat (which may have more than one layer) and non-straight (i.e., with one layer having potentially more than one transactions).

- *Corresponding Author: Yan Chen*
- *Ling Jin and Di Zhang are with the College of Computer Science and Technology, Zhejiang University, Hangzhou 310027. E-mail: ljin1995@zju.edu.cn, and diz1997@zju.edu.cn.*
- *Yinzhi Cao is with the Department of Computer Science, Johns Hopkins University, Baltimore, MD, USA, 21218. E-mail: yinzhi.cao@jhu.edu.*
- *Yan Chen and Simone Campanoni are with the Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL, USA, 60208. E-mail: ychen@northwestern.edu, and simonec@eecs.northwestern.edu.*

Once a PTS is generated, EXGEN further generates an attack contract as the exploit via two phases. First, EXGEN generates an attack contract with symbolic values. Specifically, EXGEN generates a totally-ordered sequence of transactions and thus function calls by traversing the Hasse diagram of PTS layer by layer. The generated totally-ordered sequence containing all the partial order relations in PTS forms into an attack contract with symbolic values. Second, EXGEN concretizes the generated, symbolic contract with values. Specifically, EXGEN symbolically executes the generated attack contract together with the target contract and concrete values crawled from the public blockchain. The symbolic execution extracts constraints along a certain path leading to the vulnerable sink function and tries to solve all the constraints with a solver. If EXGEN cannot find a solution after exhausting all possible execution paths, EXGEN will generate another PTS following other transactional dependencies. When EXGEN finds a solution, EXGEN deploys the attack contract in a private blockchain with states crawled from the public to verify its exploitability.

We implemented a prototype of EXGEN on two popular blockchain platforms, EOS and Ethereum. Our evaluation on EOS reveals 24 zero-day vulnerable contracts with 50 zero-day vulnerabilities, which includes a high-value EOS game contract with 10,359 transactions on 147,964 EOS coins equaling to $835,997. We also evaluated the effectiveness of EXGEN in generating exploits: The results show that EXGEN successfully exploit 1,258/1,399 (89.9%) Ethereum and 126/130 (96.9%) EOS vulnerabilities. As a comparison, the state-of-the-art approach, namely teEther, only generates exploits for 17/76 (22.4%) Ethereum vulnerabilities in scope of their paper.

We make the following contributions.

- We designed and implemented an open-source, automated exploit generation framework, called EXGEN.
- We designed Partially-ordered Transactional Set (PTS) for EXGEN to generate a symbolic contract for exploiting vulnerabilities that require multiple transactions.
- EXGEN reports 24 EOS contracts with zero-day vulnerabilities, which includes a high-value contract with 10,359 transactions. We reported all findings to contract developers if we can find them but have not heard from them yet.
- We evaluated EXGEN and showed that EXGEN can generate exploits for Ethereum and EOS contracts.

2 OVERVIEW

In this section, we start from a motivating example and then describe our threat model.

2.1 A Motivating Example

In this subsection, we illustrate an EOS smart contract with an integer overflow vulnerability as a motivating example in Figure 1. The contract allows an employer like a company to transfer equivalent amount of tokens from the employer's account to several employees via a `batchTransfer` function (Line 22). This function has integer overflow vulnerabilities on the addition and multiplication at Line 28. We focus on the multiplication in this motivating example:

```
1  class [[eosio::contract("IntflowSampleEOS")]]
       IntflowSampleEOS : public eosio::contract {
2    private:
3      balance_table  balances;
4      std::vector<name> employees;
5      uint64_t amt;
6      uint64_t rate;
7      uint64_t total;
8    public:
9      [[eosio::action]] void initEmployees(name accounts
         []) {
10       int length = sizeof(accounts) / sizeof(name);
11       total = 0;
12       for (int i = 0; i < length; ++i)
13         employees.push_back(accounts[i]);
14     }
15     [[eosio::action]] void setAmount(uint64_t amount) {
16       if (employees.size() > 0)
17         amt = amount;
18     }
19     [[eosio::action]] void setExRate(uint64_t exrate) {
20       rate = exrate;
21     }
22     [[eosio::action]] void batchTransfer(name from) {
23       require_auth(from);
24       require_recipient(from);
25       check(amt > 0, "must transfer positive amount");
26       uint64_t count = employees.size();
27       // Overflow
28       total += amt * count * rate;
29       // Omit some code here.
30     }
31 };
```

Fig. 1: A Motivating EOS Example Simplified from a Real-world Integer Overflow Vulnerability

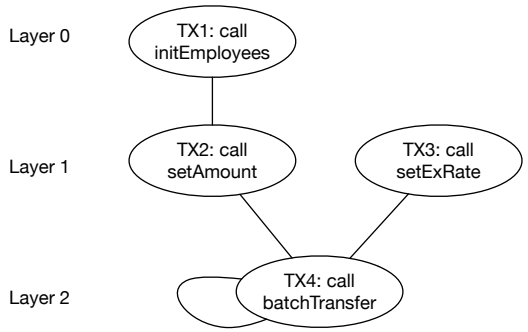


Fig. 2: Hasse Diagram of Partially-ordered Transactional Set of the Integer Overflow Vulnerability in Figure 1. There are two totally ordered transaction (TX) chains: TX1–TX2–TX4 and TX3–TX4.

Specifically, the multiplication of three variables (controlled by an adversary) could be larger than the maximum integer, thus triggering an overflow.

We now describe how EXGEN generates an attack contract in Figure 3 as an exploit to the vulnerability of the contract in Figure 1. The generation procedure of the exploit has two phases. First, in Phase One, EXGEN generates a symbolic attack contract like Figure 3 without concrete values. Specifically, EXGEN performs a backward call graph analysis from the vulnerability location (Line 28) to find an entry function, e.g., a public method, of the target contract, which is `batchTransfer` (Line 22) in Figure 1. Then, EXGEN finds all the variables that do not have a data dependency with the entry function's parameters—those variables are `employees` at Line 26, `amt` at Line 28, `rate` at Line 28, and `total` at Line 28. Next, EXGEN follows the backward dataflow to find the definition locations of those

```
1  // accounts =
       {"0","1","2","3","4","5","6","7","8","9"};
2  class AttackContract {
3    [[eosio::action]]
4    void main() {
5      initEmployees(<<SYMBOL1:accounts>>);  // TX1
6      setAmount(<<SYMBOL2:MAXINT/60>>); // TX2
7      setExRate(<<SYMBOL3:6>>); // TX3
8      for (i1=0; i1 < <<SYMBOL4:1>>; i1++)
9        batchTransfer(<<SYMBOL5:accounts[0]>>); // TX4
10   }
11 };
```

Fig. 3: A Symbolic Attack Contract with Concretized Values to Exploit the Vulnerable EOS Contract in Figure 1 (<<SYMBOLX: VALUE>> means the symbols and corresponding values.)

variables, i.e., Line 13, Line 17, Line 20, and Line 28.

Similar to the previous analysis on the vulnerability location, EXGEN finds entry functions of those variable definition locations and adds a binary relation $\leq$ between those functions and the previous entry function related to the vulnerable location in the Partially-ordered Transactional Set (PTS). That is, EXGEN adds `TX4:batchTransfer` $\leq$ `TX3:setExRate`, `TX4:batchTransfer` $\leq$ `TX2:setAmount`, `TX4:batchTransfer` $\leq$ `TX1:initEmployees`, and `TX4:batchTransfer` $\leq$ `TX4:batchTransfer` to PTS. The process is repeated until no more variables need to be initiated—for this specific example, EXGEN will also add `TX2:setAmount` $\leq$ `TX1:initEmployees` to PTS and the final Hasse diagram is shown in Figure 2. Then, EXGEN condenses the PTS following the target contract's call graph—i.e., if one function calls another and both functions are in PTS, EXGEN will merge those two functions and move them to a lower layer of the Hasse diagram. Next, EXGEN traverses the PTS layer by layer to generate a symbolic attack contract in Figure 3 with a specific transaction sequence, i.e., TX1–TX2–TX3–TX4 derived from the Hasse diagram in Figure 2. EXGEN repeats TX4 for multiple times with a symbol value, because TX4 also has a relation with itself, i.e., `total` at Line 28 also depends on itself.

It is worth noting that prior work, e.g., teEther, cannot analyze such complex data dependencies among transactions, because of the non-straight structure. Particularly, teEther can only extract two paths, which are TX1-TX2-TX4 and TX3-TX4, but cannot combine these two paths together and generate an exploit.

Second, in Phase Two, EXGEN concretizes the symbolic attack contract in Figure 3 with values. Specifically, EXGEN symbolically executes the generated attack contract and extracts constraints along a certain path. The extracted constraints corresponding to an exploitable path in this example are `employees.size()>0` (Line 16), `require_auth(from)` (Line 23), and `amt>0` (Line 25). EXGEN also adds a special constraint that overflows the `total` variable at Line 28, i.e., `total==MAXINT`. Then, EXGEN asks a constraint solver to give a solution, which concretizes all the symbols in the function parameters of all the transactions in Figure 2. Therefore, EXGEN generates a concrete attack contract with values in Figure 3. Note that values in Figure 3 belong to just one possible solution and may differ in different runs.

2.2 Threat Model

In this subsection, we describe EXGEN's threat model, which assumes that the contract developer is benign but the developed contract could be vulnerable. The victim could be other participants or the platform like EOS and Ethereum; The adversary is one of the participants of the vulnerable contract, who can commit transactions to the blockchain or deploy any contracts him or herself. We use our motivating example in Figure 1 for an explanation. The adversary is the employee, a special contract participant, because of access controls in Lines 23–24. The integer overflow attack consequence is as follows. The adversary steals coins from the EOS network, because the total number of coins is unbalanced after transfer. Particularly, some employers, possibly controlled by the adversary, obtain extra coins, but the employee does not lose corresponding amounts.

We consider a smart contract vulnerability in-scope if the vulnerability has an *explicit sink function or statement*. We now list several in-scope vulnerabilities below. Our vulnerability definitions follow prior work [11].

- Suicidal. An adversary exploits an unprotected interface to destruct a victim contract. The vulnerable sink statement is related to the `selfdestruct` instruction. The adversary is any contract located on the blockchain.
- Call injection. An adversary calls a sensitive function, e.g., ownership transfer, of a victim contract. The vulnerable sink statements are `call`, `callcode`, and `delegatecall` instructions. The adversary is any contract located on the blockchain.
- Arbitrary value transfer (e.g., Reentrancy). An adversary transfers either tokens or ether from a victim contract; if the transfer is repeated, we call it a reentrancy attack. The vulnerable sink function here is the call of `call.value`. The adversary is any contract located on the blockchain.
- Integer overflow/underflow. An adversary overflows or underflows an integer variable with a fixed length. The vulnerable sink statements are `add`, `sub`, and `mul` instructions. If the vulnerability locates in A function that any contracts can invoke, the adversary is also any contract located on the blockchain, which does not have access to privileged functions, e.g., function with an `onlyOwner` modifier. If the vulnerability locates in a function that special contract(s) can invoke, the adversary is also the special contract(s) located on the blockchain.

That said, if a vulnerability does not have an explicit sink function or statement as described above, they are out of the scope of EXGEN. For example, the definition of Prodigal contracts [6] is only in-scope when the sink functions fall into the above list. Then, transaction-ordering dependence (TOD) is out of scope because it is triggered as long as the order of transactions changes instead of any explicit statement or functions. Similarly, denial of service (DoS) is out of scope because many statements or functions may be involved in DoS.

It is worth noting that our threat model is broader than the state-of-the-art approach, i.e., teEther [15]. The first three vulnerability types (i.e., suicidal, call injection, and

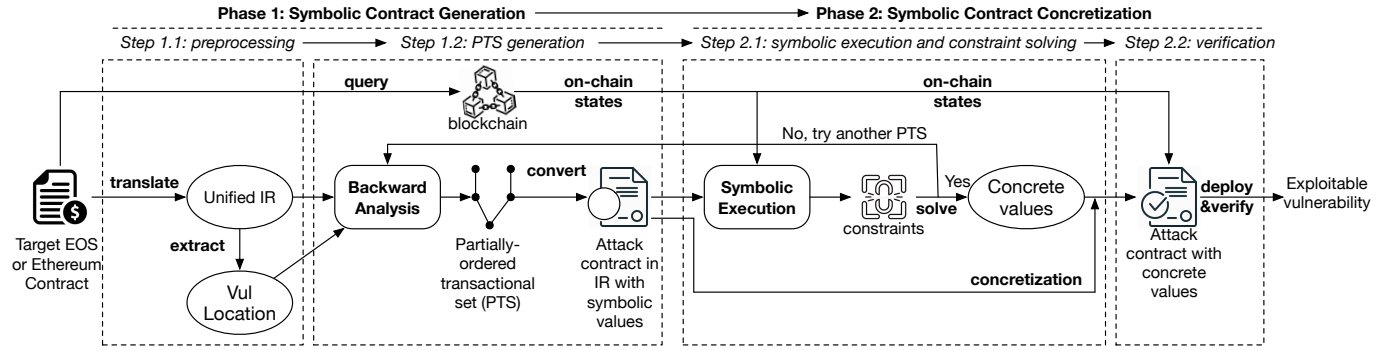


Fig. 4: EXGEN Architecture

arbitrary value transfer) entirely cover all vulnerabilities (i.e., both "direct value transfer" and "code injection" types) defined in the teEther paper. Specifically, "direct value transfer" is covered in both suicidal and arbitrary value transfer and "code injection" in call injection. Other than those covered by teEther, EXGEN also covers `call` in call injection, reentrancy and integer overflow. teEther cannot generate exploit for reentrancy because it does not handle interactions between contracts; it cannot generate exploit for integer overflow/underflow vulnerability because it does not consider constraints extracted from dataflows, e.g., an input that overflows a fixed-length variable.

3 SYSTEM DESIGN

In this section, we describe the design of our automatic exploit generation for smart contract vulnerabilities.

3.1 Architecture

Figure 4 shows the overall architecture of EXGEN with two major phases and four detailed steps. The first phase with two steps is to generate a symbolic attack contract when given a target, vulnerable smart contract on different platforms, such as Ethereum and EOS. Specifically, Step 1.1 in Figure 4 is to preprocess the target contract via translating it to a unified intermediate representation (IR) or particularly LLVM. This step depends on the target contract in different formats, e.g., Solidity, C++ or wasm binary. Then, Step 1.2 is to perform a backward analysis including call graph and dataflow and generate Partially-ordered Transactional Set (PTS). One PTS contains all the transactions that prepare for the vulnerable states and trigger the final smart contract vulnerability. EXGEN traverses the PTS to generate a symbolic attack contract.

The second phase with two steps as well is to concretize the symbolic contract with values. Specifically, Step 2.1 in Figure 4 is to symbolically execute the attack contract together with the target contract and online states crawled from the blockchain. Then, EXGEN extracts all the constraints, such as conditions in `if` statements, along one execution path leading to the vulnerable location. For dataflow related vulnerabilities such as integer over- or underflows, EXGEN also adds an additional constraint to trigger the vulnerability at the vulnerable location. Then, EXGEN asks a solver, such as z3, to give a solution based on all the collected constraints. EXGEN will try all possible execution paths for a solution.

TABLE 1: Translation of Solidity Code to LLVM IR

AST node	LLVM feature
ContractDefinition	Module
FunctionDefinition	FunctionType, Function
EventDefinition	FunctionType, Function
Block	Block
VariableDeclarationStatement	IRBuilder.store
VariableDeclaration	GlobalVariable, IRBuilder.alloca
Mapping	LiteralStructType
StructDefinition	LiteralStructType
ArrayTypeName	ArrayType
EnumDefinition	GlobalVariable
EnumValue	Constant
Literal	Constant
Assignment	IRBuilder.load/store/extract_value, IRBuilder.insert_value/BinaryOperation
IfStatement	Block, IRBuilder.branch/cbranch
WhileStatement	Block, IRBuilder.cbranch
ForStatement	Block, IRBuilder.cbranch
FunctionCall	IRBuilder.call, Function
Continue/Break	IRBuilder.branch
Return/Throw	IRBuilder.ret

If EXGEN found a solution for a particular symbolic contract, Step 2.2 in EXGEN is to verify the found exploit, i.e., an attack contract with concrete values by deploying it on a private chain with states crawled from public. Otherwise, if EXGEN cannot solve one symbolic contract with a particular PTS, EXGEN will try another PTS and symbolic attack contract in Step 1.2 for constraint extraction and solving until no more symbolic contracts are available.

3.2 Preprocessing: Code or Binary Translation

In this subsection, we describe how to translate either source code or binary to an IR form, particularly LLVM.

3.2.1 *Solidity→IR*

The overall translation as shown in Table 1 works as follows: EXGEN parses Solidity code into abstract syntax tree (AST) and then converts each AST node to corresponding LLVM IRs. EXGEN handles two types of information in Solidity AST, which are types and keywords. First, types are either mapped to a corresponding one, like `bool` in Solidity vs. `int1` in LLVM and `byte` in Solidity vs. `int8` in LLVM or implemented with native LLVM data structures, e.g., a dynamic array. Second, keywords are converted according to certain rules, e.g., an `if` statement to Block and IRBuilder.branch/cbranch in LLVM. We now describe some special cases in the translation:

- State and local variables. EXGEN represents state variables, i.e., those stored on the blockchain, as global in

LLVM and dynamically allocates memories for local variables, i.e., those are destroyed after function return.

- Function overloading. EXGEN adds a hash value of the parameter type list to the function identifier in the module and symbol table of LLVM to differentiate overloaded functions.
- Function modifiers. EXGEN translates a modifier's wrapper as an `if-then-else` statement: If the `if` condition is satisfied, EXGEN continues the function execution; otherwise, EXGEN exits the execution and returns a value. Take the `onlyOwner` modifier below for example.

```
modifier onlyOwner {
    require(msg.sender == owner);
    _;
}
function close() public onlyOwner {
}
```

The modifier is translated to the pseudocode below with an `if` statement.

```
if (msg.sender == owner) {
    close();
}
```

- Built-in EVM functions. EXGEN simulates the behavior of the corresponding built-in EVM functions; if the function is related to blockchain, EXGEN will return a pseudo value. For example, EXGEN fetches current system time and returns it in 256-bit integer when calling block.timestamp. For another example, EXGEN returns zero as default for blockchain related built-in functions such as block.number and blockhash. Note that such a zero value does not affect EXGEN's performance, because EXGEN will later on replace the value with status crawled from the chain state when generating symbolic attack contracts.
- Ether units. EXGEN converts floating-point transactions into lower monetary units, and generates LLVM bitcode using integer arithmetic, e.g., EXGEN converts 1/2 Ether to 500 Finney.
- `selfdestruct` and `throw`. EXGEN translates `selfdestruct` and `throw` to program exits.

3.2.2 EOS→IR

In this part, we describe how to translate EOS contracts to LLVM IR. The translation depends on whether source code of the contract is available.

- Open-source: C++→IR. The translation from C++ in an EOS contract to LLVM IR is intuitive because the EOS official compiler `eosio.cdt` is based on clang, which directly supports LLVM IRs internally. We skip the details of translating C++ to LLVM because this is a well-studied problem in the literature [16].
- Closed-source: wasm binary→IR. The translation from wasm binary to LLVM IR has two steps. First, EXGEN converts wasm to C code via transpiling: Note that such transpiling is a mature technique according to prior work [17]. Second, EXGEN translates C code to LLVM IR according to the same procedure in C++→IR.

3.3 Partially-ordered Transactional Set (PTS) Generation

In this subsection, we describe how to generate Partially-ordered Transactional Set (PTS) for a given vulnerability and then convert PTS to a symbolic contract.

3.3.1 PTS Definition

We now introduce the definition of PTS in Definition 1.

Definition 1 (PTS). A partially-ordered transactional set (PTS) is defined as a set of transactions together with a binary relation ($\leq$) indicating the order of transactions that appear in time axis.

We would like to note that the binary relation in PTS follows a partial order, i.e., being reflexive, antisymmetric, and transitive, because transactions are recorded on blockchain with exact timestamps. Furthermore, those transactions are not totally ordered as the order of some transactions, e.g., TX2 and TX3 in Figure 2, is unranked on the time axis. In other words, only a transaction that prepares states for another transaction, e.g., TX1 and TX2 in Figure 2, has a partial order relation with and needs to come before the latter.

3.3.2 Initial PTS Construction via Backward Analysis

We describe how to construct an initial set of PTS via backward analysis. This construction has three steps: (i) backward call graph analysis to discover transactions' entry points, (ii) backward dataflow analysis to discover additional transactions having data dependencies with existing ones, and (iii) repeat of (i) and (ii) until no more new transactions are needed.

Let us describe the steps below. First, EXGEN starts from a particular IR instruction, e.g., an integer overflow instruction in the beginning or a def instruction of a particular variable, follows the call graph to find the entry function of a transaction, e.g., a public function. Second, EXGEN will extract all the external variables unrelated to the function parameters following the call chain. Then, EXGEN follows the use-def chain to find the instructions that define the aforementioned external variables. Note that there might be multiple definitions of a certain use of external variables, which lead to multiple PTSes. Lastly, EXGEN repeats the first step to find another entry function and finds external variables during the process as well.

3.3.3 PTS Condensing via Call Graph

Now we describe how to condense PTS—we need to condense PTS because some entry functions in PTS have call relations and thus there is no need for two transactions. The process is as follows. First, EXGEN goes through PTS and replaces one function call with its direct caller or ancestor for a transaction if the latter is also in PTS. The original partial order is kept but replaced with the new transaction. Second, EXGEN merges identical transactions in PTS and moves the transaction to the largest layer in the Hasse diagram of PTS.

3.3.4 PTS→Symbolic Contract

We now present how to convert PTS to a symbolic contract. EXGEN represents PTS in a Hasse diagram with layers. EXGEN starts from the first layer, traverses all the transactions in the layer by creating a totally-ordered set, and then goes to the next layer. The totally-ordered set indicates the sequence of transactions: EXGEN creates a contract to invoke all the entry functions of each transaction with symbols as the parameters.

Note that if a contract also has a relation with itself in the diagram (e.g., TX4 in Figure 2), EXGEN will handle it via two ways. If the transaction is the last one targeting a reentrancy vulnerability, EXGEN creates a fallback function with that transaction because the fallback function will be called automatically in a loop. Otherwise, EXGEN creates a `for` loop that executes for unknown times, i.e., the loop number is a symbolic value.

3.4 Symbolic Execution and Constraint Solving

In this subsection, we describe how EXGEN collects constraints and concretizes symbols to trigger vulnerabilities when symbolically executing the attack contract with symbolic values and subsequently the target contract. There are four steps: (i) initiating storage values with states crawled from public blockchain, (ii) selection of a certain symbolic attack contract with a PTS and extraction of constraints along the execution of the PTS, (iii) generating constraints related to the vulnerability, and (iv) solving the extracted constraints and trying other symbolic attack contracts if no solution is found.

3.4.1 State Initialization via Crawling from Public Blockchain

The first step is to initialize storage values in symbolic execution with those crawled from public blockchain in real-time. Specifically, because storage values are translated as global variables in the IR form, EXGEN will assign values directly in the beginning of the symbolic execution as part of the IR code. It is worth noting that although storage values are initialized with those from public chain, they can be changed later by transactions in the attack contract.

3.4.2 Symbolic Execution and Constraint Extraction

The second step is to symbolically execute the attack contract as well as the target contract. EXGEN starts from the first transaction and follows the order in the attack contract in the symbolic execution. EXGEN extracts all the conditions in the branching statements, such as `if` and `switch`, and specifies the constraint based on the branching condition in the execution path. We now look at some examples.

- `if` statement. EXGEN specifies a constraint with the `if` condition as true if the symbolic execution goes to the `if` branch, and as false for the `else` branch.
- `switch` statement. EXGEN specifies a constraint with the `switch` condition as the executed `case` condition.
- `while` or `for` loop. If the loop body is executed, EXGEN specifies the condition in either `while` or `for` statement as true.

This symbolic execution stops at the vulnerability statement with an related instruction, such as `add`, `mul`, `sub`, `call.value`, `selfdestruct`, `call`, `callcode` and `delegatecall`. Then, EXGEN also produces an execution path depending on which branch that EXGEN takes when extracting each constraint.

3.4.3 Vulnerability-specific Execution

The third step is to perform vulnerability-specific symbolic execution at the vulnerable statement. There are two cases. First, there exists a dataflow constraint to exploit the vulnerability and EXGEN will construct a constraint related to the vulnerability. In this case, EXGEN symbolically executes the vulnerable instruction and constructs a constraint based on the vulnerability type if there exists a dataflow constraint. We list two vulnerability types below.

- Integer Overflow. EXGEN specifies that the addition or the multiplication result exceeds the maximum value of the corresponding integer type plus one, e.g., 2^{256} for uint256.
- Integer Underflow. EXGEN specifies that the subtraction result is less than zero, which causes an underflow for an unsigned integer, a common type used in digital assets like account balance in bank contracts and scores or points in gaming contracts.

Second, only control-flow constraints are enough for exploitation. In this case, EXGEN checks whether the vulnerability exists. That is, EXGEN checks whether the caller to `call.value` is a public function for reentrancy; EXGEN also checks whether the parameter to `selfdestruct`, `call`, `callcode` and `delegatecall` can be externally controlled for suicidal and call injection.

3.4.4 Constraint Solving

The last step is to solve the extracted or constructed constraints from previous two steps using a constraint solver and produce concrete values for all the symbols. If EXGEN cannot find a solution, EXGEN will try another symbolic attack contract and extract new constraints. EXGEN will continuously try all the possible symbolic contracts until a solution is found.

3.5 Exploit Verification

In this subsection, we describe the process of concretizing the attack contract and verifying whether it can exploit the target vulnerability on a private chain with states crawled from the public. Note that due to ethics concerns, we cannot directly deploy the attack contract on the public chain for verification.

3.5.1 Attack Contract Concretization

We now introduce how to extract the concrete values solved by symbolic execution and generate an attack contract. When EXGEN determines that the extracted constraints from executing a symbolic attack contract can be solved, it will assign the concrete values to the corresponding transactions' parameters according to the totally-ordered set.

- Solidity contract. EXGEN generates a Solidity contract and then compiles the Solidity code to the bytecode for deployment.

- Open-source EOS contract. EXGEN generates a C++ contract and then compiles the contract to wasm using `eosio.cdt`.
- Closed-source EOS contract. EXGEN generates a contract in wasm directly.

3.5.2 Exploit Verification

EXGEN verifies whether the generated attack contract can exploit the vulnerability via two steps. First, EXGEN deploys the contract on a private blockchain with values crawled from the public chain. Second, EXGEN runs a modified version of Ethereum Virtual Machine (EVM) or EOS Virtual Machine (EOS-VM) with the contract to determine the exploitability. Specifically, EXGEN instruments all sink instructions to determine the runtime states.

- `add` and `mul`. EXGEN first determines whether the arithmetic result of the left and right source operands is less than the destination operand in the modified EVM or EOS-VM. Then, EXGEN checks whether state variables stored on the blockchain are changed: If so, EXGEN considers the exploit as successful.
- `sub`. EXGEN determine whether the left operand (minuend) is less than the right operand (subtrahend) in the modified EVM or EOS-VM. Then, EXGEN follows the above to check state variables and determines the exploit as successful.
- `selfdestruct` (or named `suicide` in old Solidity versions). EXGEN determines whether the balance of the attacker's address has increased. If so, EXGEN considers the exploit as successful. Note that the receiving address of ether transfer is a required parameter of the `selfdestruct` instruction.
- `call.value`. EXGEN determines whether the entry function, which contains the invocation to `call.value` instruction, is called in multiple transactions. Then, EXGEN checks whether the balance of the attacker's address has increased. If so, EXGEN considers the exploit as successful.
- `call`, `callcode` and `delegatecall`. EXGEN verifies call injection by injecting a `selfdestruct` and the verification boils down to the verification of a suicidal.

4 IMPLEMENTATION

In this section, we describe our prototype implementation of EXGEN, which has 3,400 Lines of Python and 4,600 Lines of C++ Code. Our implementation is open-source and available at Google Drive [18]. We now describe how each step of EXGEN is implemented.

- Preprocessing. Our Solidity→LLVM translator implementation with 1,478 lines of Python code is based on ANTLR4 grammar and llvmlite [19] 0.31.0, a third-party Python library. Our binary EOS→LLVM adopts the wasm2c tool in the WebAssembly Binary Toolkit (WABT) to convert wasm to C code and then use the clang compiler with the -emit-llvm option to compile the header and source code files to LLVM IRs. Our source-code EOS→LLVM adopts `eosio.cdt` with 245 lines of modifications to the libraries.
- Backward Analysis and PTS Generation. Our implementation of backward analysis and PTS generation is a customized LLVM analysis module with 1,324 lines of Python code. This module of our prototype EXGEN implementation analyzes functions, basic blocks and instructions to construct a call graph and a dataflow graph represented in Python classes.
- Symbolic Execution and Constraint Solving. Our implementation of symbolic execution is based on KLEE [20], a LLVM-based symbolic virtual machine, as our symbolic executor. KLEE supports several solvers and we adopt the famous Z3 solver [21] with the `-solver-backend` option in KLEE. The total implementation of this part has 4,271 lines of C++ code, which does not include the code of KLEE itself.
- Exploit Generation and Verification. The exploit generation is a 600-Line Python script that traverses our totally-ordered set and generates a contract with concrete values. Then, the exploit verification is as follows. EXGEN deploys the contract on a private chain with several accounts and then interacts with it via APIs provided by web3.py (Ethereum client in Python version) and cleos (Official EOS client). The implementation also instruments EOS-VM (C++) and EVM (Go) with approximately 200 Lines of Code each to check the value of balance and the arithmetic instructions including `i32_add_t` and `i64_sub_t` in `base_visitor` of EOS-VM and `opAdd` or `opSub` in Go Ethereum (Geth).

5 EXPERIMENTAL SETUP AND EVALUATION BENCHMARK

In this section, we describe our experimental setup and evaluation benchmark. All our experiments are performed atop a DELL XPS9570 machine using i7-8750H CPU with six cores and 12 threads and running a virtualized 32-bit Ubuntu v16.04 with 20 GB of RAM. Our evaluation benchmark has two parts depending on the platform: one on Ethereum and the other on EOS. Table 2 shows the number of exploitable vulnerabilities and their breakdown based on types on each platform. We now describe them below.

Ethereum Benchmark. Our Ethereum benchmark has 562 real-world vulnerable contracts with 1,399 vulnerabilities and 798 real-world safe contracts. We obtain the vulnerable contracts from datasets provided by prior works, namely HuangGai [22], ContractFuzzer [14], Zeus [5] and Ever-evolving Game (EEG) [11]. Table 2 shows the breakdown of vulnerabilities from different sources and we also describe them below:

- HuangGai. This dataset contains 100 integer overflow/underflow vulnerabilities, 30 reentrancy vulnerabilities and 32 suicidal vulnerabilities from 101 real-world contracts.
- ContractFuzzer. This dataset consists of 41 real-world contracts, covering two types of vulnerability. Specifically, this dataset contains 31 call injection and 18 reentrancy vulnerabilities.
- Zeus. This dataset contains 31 contracts with reentrancy and 458 with integer overflows. Our manual verification confirms 39 reentrancy vulnerabilities from 26 contracts and 1,029 integer overflow/underflow from 365 contracts. The rest contracts are false positives of Zeus

TABLE 2: Evaluation Benchmark on Ethereum and EOS.

vulnerability type	instructions	Ethereum					EOS		
		HuangGai	ContractFuzzer	Zeus	EEG	Total	Real-world	Synthetic	Total
Overflow	`add`	54	-	624	50	728	30	28	58
	`mul`	7	-	209	14	230	12	5	17
Underflow	`sub`	39	-	196	33	268	9	46	55
Arb. value transfer (e.g., Reentrancy)	`call.value`	30	18	39	10	97	-	-	-
Suicidal	`selfdestruct`	32	-	-	13	45	-	-	-
Call Injection	`call callcode delegatecall`	-	31	-	-	31	-	-	-
Total		162	49	1,068	120	1,399	51	79	130

because they are unexploitable: This is also the necessity of EXGEN in finding and solving constraints.

- Ever-evolving Game (EEG). This dataset contains 97 integer overflow/underflow vulnerabilities, 10 arbitrary value transfer vulnerabilities and 13 suicidal vulnerabilities from 35 real-world contracts.

We obtain our safe contracts by both pre-filtering via existing works and manual verification. Specifically, we first obtain 66,103 untagged contracts from HuangGai [22] and scan them with Slither [23], which reports 803 contracts as safe. Next, we use Securify [24] for a further scanning and then manually verify the rest, which in the end produces 798 safe contracts.

EOS (C++ & wasm) Benchmark. Our EOS benchmark has 162 real-world, unlabelled contracts crawled from eospark.com on April 2020. Our manual verification reveals 51 real-world zero-day exploitable vulnerabilities from 24 contracts in those contracts. Additionally, we also asked a graduate student (independent of the paper authors) to generate 79 synthetic vulnerabilities via removing assert and check statements from 36 contracts. The generation takes the student 64 hours. Note that sometimes, he needs rewrite the contract code in old version based on the latest grammar in 1.6, such as removing the typedefs of account_name and renaming checksum() or public_key() APIs. Then, the student manually wrote an exploit for each generated EOS contract and deployed the contract using `cleos` to verify the exploit. To summarize it, details of the real-world and synthetic vulnerabilities and their break-downs are shown in Table 2.

6 EVALUATION

We describe the research questions (RQs) to answer in our evaluation.

- RQ1 [Effectiveness]: What is the success rate of EXGEN in generating exploits for verified, vulnerable contracts when comparing with the state-of-the-art approach?
- RQ2 [Zero-day and Unexploitable]: Can EXGEN find and exploit zero-day vulnerabilities?
- RQ3 [Overhead and Scalability]: What is the overhead and scalability of EXGEN in exploit generation?
- RQ4 [Solvable Symbolic Contracts]: How many symbolic attack contracts will EXGEN produce and how many of them are solvable?
- RQ5 [Constraints]: How many constraints will EXGEN produce in analyzing solvable symbolic attack contract?

6.1 RQ1: Effectiveness

In this subsection, we answer the question of how effective EXGEN is in generating exploits for vulnerable contracts in our benchmarks. The evaluation methodology is as follows. We use EXGEN to exploit each vulnerability in the vulnerable contract and then verify the generated exploit using an offline private chain with public chain states. There are two things worth noting here. First, EXGEN generates multiple exploits if the contract has more than one vulnerability. Second, as a sanity check, we also run EXGEN on Ethereum and EOS safe contracts and the results show that none of the safe contracts is exploitable.

Table 3 shows the success rate of EXGEN in exploiting existing Ethereum (Solidity) vulnerabilities. EXGEN exploits 1,258 out of 1,399 vulnerabilities, i.e., 488 out of 562 vulnerable contracts. 118 out of 141 failed cases are due to our translator implementation: Currently, the Solidity→LLVM translator has limited support on heterogeneous `struct` type, e.g., `int`, `mapping`, and `array` in the same `struct` type. The other 23 failed cases are that the symbolic execution cannot generate PTS due to timeout.

As a comparison, teEther [15] only successfully exploits 17 out of 76 vulnerabilities that it supports. Such a low success rate is two-fold. First, teEther does not finish analyzing many contracts after one hour or cannot generate vulnerability spanning across multiple contracts. The total of such case takes about 60% of unexploited contracts. Second, around 20% of contracts need transactions with non-straight dependencies for exploitation.

We also breakdown the success rate on Ethereum by different datasets and show them in Table 4. The numbers are most consistent across different datasets. It is interesting that the success rate on EEG, which contains mostly vulnerable contracts with real-world attacks, is very high. That is, EXGEN is very effective in generating exploits that are used by real-world attackers.

Apart from Ethereum, Table 5 shows the success rate of EXGEN in exploiting existing EOS (C++/wasm) vulnerabilities. EXGEN successfully exploits 122 out of 130 vulnerabilities in 55 open-source contracts and 120 out of 130 vulnerabilities in 53 closed-source EOS contracts. Interestingly, when we combine them together, EXGEN can exploit 126 out of 130 vulnerabilities in 58 EOS contracts, because the failing reasons are different on open- and closed-source contracts. The major reason of failing to exploit open-source contracts is that some types, such as vectors, are unsupported in

TABLE 3: [RQ1] Exploitation Results for Ethereum (Solidity) Vulnerabilities of EXGEN and Comparison with teEther.

vulnerability type	instructions	teEther			EXGEN		
		# exploitable	# exploited	success rate	# exploitable	# exploited	success rate
Overflow	`add`	-	-	-	728	669	91.9%
	`mul`	-	-	-	230	201	87.4%
Underflow	`sub`	-	-	-	268	228	85.1%
Arb. val. trans.	`call.value`	-	-	-	97	88	90.7%
Suicidal	`selfdestruct`	45	10	22.2%	45	44	97.8%
Call Injection	`call` or `callcode` or `delegatecall`	31	7	22.6%	31	28	90.3%
Total		76	17	22.4%	1,399	1,258	89.9%

TABLE 4: [RQ1] Breakdown of Exploit Generation Effectiveness by Different Datasets on Ethereum (Solidity).

	# exploitable	# exploited	success rate
HuangGai	162	159	98.1%
ContractFuzzer	49	44	89.8%
Zeus	1,068	941	88.1%
EEG	120	114	95.0%
Total	1,399	1,258	89.9%

the implementation; the major reason of failing to exploit closed-source contracts is timeout. Some of unsupported types are converted to a lower-level representation if converted to wasm and thus can be handled by EXGEN.

6.1.1 A Case Study

In this part, we introduce a case study on an Ethereum contract that can be detected by EXGEN but not teEther. The Ethereum contract, called Ethsplit [25], is a ERC token with a reentrancy vulnerability. Figure 5 shows the source code of the Ethsplit contract with two reentrancy vulnerabilities at Line 6 and 12 respectively. Here is how EXGEN generates exploit for the second reentrancy. The PTS only has one transaction with a self-dependency and therefore EXGEN creates an `exploit` and another fallback function. Then, EXGEN extracts two constraints, which are `amIOnTheFork.forked() == false` and `fee == msg.value/100` based on the `if` statement branch. Next, EXGEN solves the value of `fee` as 0.01 when setting `msg.value` as 1 and generates a concrete attack contract at Line 18-26. Lastly, EXGEN fetches on-chain state, in which the stored value `amIOnTheFork.forked` returns false, and verifies that the exploit works.

[RQ1] Take-away: EXGEN significantly outperforms state-of-the-art exploit generation approach, namely teEther, in (i) success rate of exploitation (89.9% vs. 22.4%), (ii) number of vulnerability types (5 vs. 2), and (iii) number of platforms (Ethereum + EOS vs. Ethereum only).

```
1  /* Source code of the vulnerable contract*/
2  contract Ethsplit {
3    function split(address ethAddress, address etcAddress
         ) {
4      if (amIOnTheFork.forked()) {
5        // if on the forked chain send ETH to ethAddress
6        ethAddress.call.value(msg.value)();
7      }
8      else {
9        // send ETC to etcAddress less fee
10       uint fee = msg.value/100;
11       fees.call.value(fee);
12       etcAddress.call.value(msg.value-fee)();
13     }
14   }
15   ...
16 }
17 /* Attack contract targeting the vulnerable contract*/
18 contract AttackContract {
19   Ethsplit target = Ethsplit(TARGET_ADDRESS);
20   function exploit() public payable {
21     target.split(<<SYMBOL1:ATTACKER_ADDRESS>>, <<
           SYMBOL2:ATTACKER_ADDRESS>>).value(<<SYMBOL3
           :1>>);
22   }
23   function() external payable {
24     target.split(<<SYMBOL1:ATTACKER_ADDRESS>>, <<
           SYMBOL2:ATTACKER_ADDRESS>>).value(<<SYMBOL3
           :1>>);
25   }
26 }
```

Fig. 5: [RQ1] An Ethsplit Ethereum Contract with a Reentrancy.

6.2 RQ2: Zero-day Vulnerabilities

In this research question, we evaluate EXGEN against 162 EOS contract for its capability in detecting and exploiting zero-day vulnerabilities. The results show that EXGEN produces 50 exploits for 24 real-world contracts with zero-day vulnerabilities. We reported all vulnerabilities and exploits to the contract developers if we can find them. It is worth noting that one contract is an EOS game contract, called Gameworldcom [26], which is vulnerable to integer overflow. The contract is very popular with 10,359 transactions totaling 147,964 EOS coins in the past.

Now, we introduce a case study on an EOS contract called Blaster [27], which creates deferred transactions for performance testing. This contract has an integer underflow vulnerability at Line 7 in Figure 6. Suppose a tester inputs a past time when testing the performance of a contract, the integer underflow vulnerability may cause an endless period before committing the transactions. Here is the exploit generation of EXGEN. The PTS only has one transaction and EXGEN extracts one constraint related to the vulnerable location. Then, EXGEN solves the value of `blast_time` as 0 considering that the value of the current timestamp is always positive. In the end, EXGEN generates an ex-

TABLE 5: [RQ1] EXGEN's Exploitation Results for EOS (C++/wasm) Vulnerabilities on EOS Benchmarks.

code	instr (vuln)	Real-world Vulnerabilities			Synthetic Vulnerabilities			Total		
		# exploitable	# exploited	success rate	# exploitable	# exploited	success rate	# exploitable	# exploited	success rate
C++	`add` (overflow)	30	28	93.3%	28	26	92.9%	58	54	93.1%
	`mul` (overflow)	12	11	91.7%	5	5	100%	17	16	94.1%
	`sub` (underflow)	9	8	88.9%	46	44	95.7%	55	52	94.6%
	Total	51	47	92.2%	79	75	94.9%	130	122	93.9%
wasm	`add` (overflow)	30	29	96.7%	28	25	89.3%	58	54	93.1%
	`mul` (overflow)	12	10	83.3%	5	5	100%	17	15	88.3%
	`sub` (underflow)	9	8	88.9%	46	43	93.5%	55	51	92.7%
	Total	51	47	92.2%	79	73	92.4%	130	120	92.3%
C++/wasm	Total	51	50	98.0%	79	76	96.2%	130	126	96.9%

```
/* Source code of the vulnerable contract */
class blaster : public contract {
  void blast( uint64_t blast_time, uint32_t start,
       uint32_t iterations ) {
    for (uint32_t i = start;i<start + iterations;i++) {
      transaction deferredTrans{};
      ...
      deferredTrans.delay_sec = blast_time -
           current_time_point().time_since_epoch().count
           ();
      deferredTrans.expiration = time_point_sec(
           current_time_point().time_since_epoch().count
           () + (60 * 60 * 24));
      uint128_t sender_id = (uint128_t(i) << 64) |
           blast_time;
      deferredTrans.send(sender_id, _self);
    }
  }
}
class AttackContract {
  [[eosio:action]]
  void main () {
    blast(<<SYMBOL1:0>>, <<SYMBOL2:ANY_VALUE>>, <<
         SYMBOL3:ANY_POSITIVE_VALUE>>);
  }
}
/* JSON objects of EOS ABI
{
  "code": "blaster",
  "action": "blast",
  "args": {
    "blast_time": 0,
    "start": 0,
    "iterations": 1
  }
} */
```

Fig. 6: [RQ2-Zero-day] An Blaster EOS Contract with an Integer Underflow.

ploit example shown in Figure 6. EOS RPC requires JSON-format data to submit the transaction, in which function name and parameter values are provided using key-value pairs. This JSON object is converted to binary through `abi_json_to_bin` and then posted to the chain by EOS RPC. Attackers can also use the `cleos` client with the command `cleos push action` to submit the transactions.

[RQ2] Take-away: On platforms with no prior detection tools on certain vulnerability types such as integer overflows (i.e., EOS), EXGEN is able to exploit zero-day vulnerabilities.

6.3 RQ3: Overhead and Scalability

In this subsection, we answer the research question of what the performance overhead of EXGEN is and whether EXGEN is scalable as the number of path increases.

6.3.1 Performance Overhead

We start from evaluating the overhead of EXGEN using the Ethereum and EOS benchmarks. Figure 7 shows the cumulative distribution function (CDF) of the performance overhead of Ethereum and EOS contracts. EXGEN is efficient in analyzing all the contracts: EXGEN analyzed 90% of Ethereum and 75% of EOS (C++) contracts within 100 seconds and 80% EOS (wasm) contracts within 300 seconds.

There are three things worth noting here. First, the analysis of EOS contract in wasm triples the time spent on the one with source code. One major reason is that EXGEN translates wasm code to C instead of C++, which needs implementations of many C++ libraries. Another reason is that many high-level semantics, such as loops, are lost in the wasm format.

Second, EXGEN takes more time to analyze EOS contracts than Ethereum. The reason is that the logics of EOS contracts are generally more complex than the ones of Ethereum and thus have more lines of code. Particularly, the execution of Ethereum contracts needs gas and the amount of gas depends on the number of instructions: Therefore, developers tend to write small, simple contracts on Ethereum.

Third, there are sudden increases on the CDF graph. For example, the CDF graph of Ethereum contracts has a big increase between 100 and 110 seconds. The reason is that many contracts in the Ethereum benchmark are derived from three popular tokens, i.e., "Proof of Weak Hands (PoWH)", "Ponzi", and "EthPyramid (EPY)", and the analysis time of those contracts are very similar. For another example, there is also an increase between 45 and 60 seconds (C++) and between 120 and 135 seconds (wasm). It is because those contracts in the EOS benchmark implement their transfer functions based on the `eosio.token` contract from the official EOSIO documentation.

Figure 7 also shows the performance overhead of teEther [15] as a comparison to EXGEN. The number of time-out contracts of EXGEN and teEther after 300 seconds is similar. The sharp increase at the beginning for teEther is due to that 95% contracts fail to compile leading to a quick error under one second.

6.3.2 Overhead Breakdown

In this part, we further break down the overhead of four Ethereum contracts in Table 6 based on the overhead of

TABLE 6: [RQ3] Breakdown of Overhead in Seconds for Ethereum and EOS Sample Contracts.

benchmarks	Contract	Total Time*	Translator	PTS Generation	Symbolic Execution	LoC
Ethereum	HmcDistributor [28]	6.854	0.071	2.477	4.306	108
	NumbersToken2 [29]	98.972	0.733	3.504	94.735	279
	Agricoin [30]	159.08	0.403	4.365	154.312	690
EOS (C++)	dacservice	38.184	2.759	3.195	32.230	126
	eosdtnutoken	59.675	3.091	4.868	51.716	306
	happyeosslot	183.361	3.891	5.935	173.535	821
EOS (wasm)	dacservice	95.612	5.87	8.503	81.239	6,436**
	eosdtnutoken	144.177	6.557	10.181	127.439	9,581
	happyeosslot	452.413	7.306	12.98	432.127	18,027

*: The total time includes the one to generate an exploit but not verification time (which depends on the number of exploits to verify).
**: The number in EOS (wasm) refers the lines of reverse engineered C code.

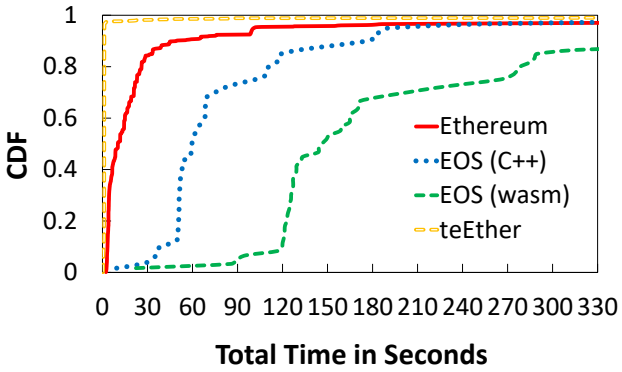


Fig. 7: [RQ3] CDF of Performance Overhead

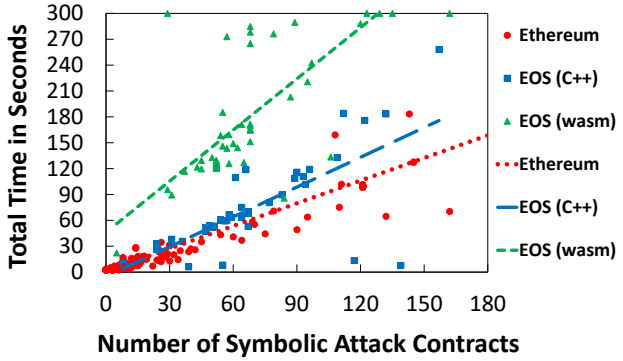


Fig. 8: [RQ3] Overhead vs. # of Symbolic Attack Contracts

different steps of EXGEN. Symbolic execution and constraint solving together are the most timing-consuming and increase as the lines of code (LoC) because EXGEN needs to explore all different possible paths. The performance overhead on PTS generation also increase linearly as LoC but in a very slow pace. The translator's performance is almost unrelated with LoC because the LoC is generally small and does not impact its performance.

6.3.3 Scalability

We evaluate the scalability of EXGEN as the number of generated symbolic attack contracts increases. Specifically, we plot a graph with the y-axis as the total analysis time and the x-axis as the number of symbolic contracts. Figure 8 shows the results: All the points in the analysis aligns well with linear lines. The lines of EOS (C++) and Ethereum are close, because the overhead in analyzing source code are similar. The line of EOS (wasm) is much higher than the one of EOS (C++) and Ethereum, because reverse engineered source code is much longer than the original.

[RQ3] Take-away: EXGEN finishes analyzing most vulnerable contracts within five minutes and is scalable to the contract size.

6.4 RQ4: Solvable Symbolic Attack Contract

In this subsection, we summarize the number of total and exploitable symbolic attack contracts using a CDF graph in Figure 9. We only draw one CDF line for EOS, because the number of generated symbolic attack contracts for open- and closed-source target EOS contracts are the same. We have several observations. First, EOS contracts tend to have more symbolic attack contracts than Ethereum, because the logics of EOS contracts are more complex than the ones of Ethereum. It is probably because EOS does not charge gas for contract execution.

Second, the solvable symbolic contract rate of EOS is lower than Ethereum: It is because EOS contracts also have more constraints, which limits the solvability of some symbolic attack contracts with a particular function invocation sequence. For example, one unsolvable case is that the symbolic contract divides a value with a number and then times the intermediate result with another number that is smaller than the former. Therefore, the final result will not be overflowed given all the constraints.

[RQ4] Take-away: The performance overhead of EXGEN increase linearly as the number of generated symbolic attack contracts.

6.5 RQ5: Constraints

In this subsection, we show a CDF graph on the number of extracted constraints when generating exploits for Ethereum and EOS contracts in Figure 10. There are two things worth noting here. First, most contracts, i.e., nearly 85% of Ethereum contracts have no more than 100 constraints and 80% of EOS (C++) ones have no more than 150 constraints. This is smaller than desktop applications, which may have thousands or tens of thousands of constraints.

Second, EOS (wasm) contracts have much more constraints than EOS (C++) because one constraint in a C++ code may be broken into multiple in the wasm format.

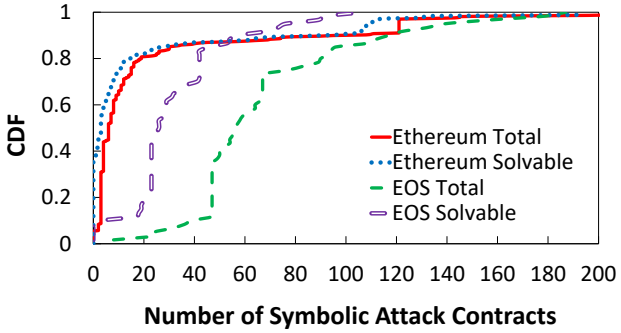


Fig. 9: [RQ4] CDF of Total and Solvable Symbolic Attack Contracts

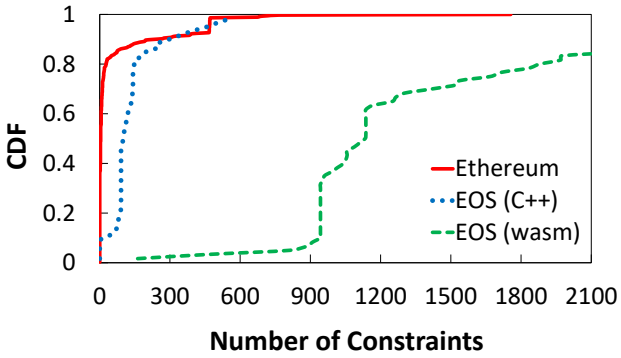


Fig. 10: [RQ5] CDF of Number of Constraints.

Take an `if` statement for example. A wasm binary and the reverse engineered C code will first store the condition, likely a comparison, into a register or a variable and then compare the stored result with zero. That is, one single constraint becomes two in wasm.

[RQ5] Take-away: The number of constraints in smart contracts is relatively small due to the natural of contract functionality.

7 Discussion

In this section, we discuss some commonly raised questions.

First, we would like to note that the prototype implementation only supports Solidity code, EOS C++ code, and EOS wasm binary. That said, our prototype of ExGen does not support smart contracts in other formats, such as Solidity binary and Java on the Hyperledger Fabric platform. We will leave them as our future work.

We also want to mention that our current implementation of Solidity→LLVM translator has some limitations. Our translator implementation of dynamic arrays and mappings are based on arrays with fixed length rather than dynamic allocation. In addition, our implementation does not fully support constant hex values. These implementation compromises do not affect exploit generation in practice—ExGen adopts a fixed, sufficiently-large length and converts hex to decimal.

Second, our prototype implementation of wasm built-in functions is restricted to the EOS benchmark. We believe that this is sufficient for our evaluations and leave a full implementation of all wasm built-in functions as a future work. Additonally, the dataflow analysis of ExGen is coarse-grained on the variable type, such as `struct` and `array` instead fine-grained on different fields of a `struct` variable. We believe that such dataflow analysis is sufficient to generate PTS in practice.

Third, we want to discuss the general problem of path explosion in symbolic execution. ExGen can find all the paths in practical smart contracts, because the logics of smart contracts are relatively simple compared with large-scale desktop or mobile applications. One reason for Ethereum contracts is that the execution of each instruction requires a gas fee and therefore developers tend to keep contracts simple.

Fourth, we discuss the impacts of compiler optimization of arithmetic operations on ExGen. In a one-sentence summary, such optimization does not have impacts on ExGen because ExGen disables optimization in the translator and thus the translated LLVM IRs preserve the original integer overflow vulnerabilities. Specifically, our translation from Solidity to LLVM is a customized implementation without optimization; we disable compiler optimization in translating EOS source code to LLVM. Furthermore, we deploy generated attack contracts on a private chain to confirm that integer overflows can be triggered.

Fifth, we discuss some control-flow condition checks that happened after vulnerability location. This is specific to smart contracts because they can abort transactions at any time. ExGen will generate exploits for such contracts but the verificaiton step (Step 2.2) will fail because of such checks.

Sixth, we discuss non-contract format exploit generated by ExGen. This is possible if some contracts check "msg.sender" with "tx.origin" and such constraints are extracted by ExGen. If so, ExGen will generate an exploit as multiple single transactions from a human account.

Lastly, we discuss how compiler-level protection of integer overflows, such as LLVM's sanitizer and GCC's built-in function, affects ExGen. We would like to note that existing contracts, particularly those on Ethereum and EOS, do not have any protection enabled. Those contracts—which cannot be changed once deployed based on the nature of blockchain—are the target of ExGen to generate exploits.

8 Related Work

In this section, we discuss related work.

Automatic Exploitation and Contract Testing. teEther [15] from Krupp et al. is probably the closest related work, which automatically generates exploits for a given vulnerable contract's bytecode. Specifically, teEther classifies four binary operations as potential injection points for exploit generation. As a comparison, teEther cannot generate exploits for integer overflows due to lack of dataflow information into arithmetic operations.

Targeting at stack overflows and format string attacks, Avgerinos et al. [31] first proposes control flow hijacking exploit generation for programs with source code. For compiled binaries, MAYHEM [32] and CRAX [33] address path selection and symbolic execution during exploit generation by implementing binary AEG features. Gollum [34] designs a grey-box approach to generate exploits focusing on heap

overflows in interpreters. FUZE [35] contributes to exploiting kernel UAF vulnerabilities by utilizing kernel fuzzing and symbolic execution. Leveraging capability modeling of OOB writes, KOOBE [36] can be applied to heap OOB write and other types of kernel vulnerabilities with the inherently multi-interaction nature of kernel. These works do not handle hash values and blockchain states in the execution environment of the EVM or the EOS. Therefore, their AEG methodology cannot be applied to smart contract vulnerabilities.

Zhang et al. [13] and Jiang et al. [14] test Ethereum smart contracts via dynamic fuzzing. Inspired by the well-known fuzzer for C programs AFL [37], sFuzz [38] proposes an adaptive strategy for improving seed selecting. SMARTIAN [39] leverages both static and dynamic analysis to deterministically identify critical transaction sequences which effectively provides feedback for guided fuzzing. EasyFlow [12] also generates transactions with large or small integers to trigger integer overflows via fuzzing. Dynamic fuzzing may not trigger the vulnerable code due to the lack of constraint solving and is complementary to exploit generation. As a comparison, EXGEN is a static analysis framework using symbolic execution to solve constraint along the exploit path.

Vulnerability Detection and Patching in Existing Smart Contract. People have also proposed many prior works in detecting vulnerabilities in smart contracts. Those work can be classified into static analysis, dynamic analysis, and formal verification. First, Oyente [4] from Luu et al. and ZEUS [5] from Kalra et al. adopt static analysis to detect many smart contract vulnerabilities. There are also other popular symbolic execution tools, such as Mythril [40], MAIAN [6], Mueller [7], Osiris [8], Lai [9] and Vandal [10]. In addition, Torres [41] used symbolic execution to observe the rising of honeypot contracts and summarized the techniques adopted by the honeypots. GASPER [42] and MadMax [43] investigate gas-focused vulnerabilities to prevent contracts from out-of-gas conditions. Zhou [44] developed Erays that reverse engineers EVM bytecode to high-level, human-readable pseudocode. As a comparison, prior works can only detect the possible existence of a vulnerability but do not generate concrete exploits.

EOSSafe [45] is a static analysis tool that detects EOS vulnerabilities including fake EOS, fake receipt, rollback and missing permission check using symbolic execution. As a comparison, vulnerabilities considered by EXGEN, e.g., integer overflow/underflow, are out of scope of EOSSafe. Moreover, EOSSafe, unlike EXGEN, only detects vulnerabilities but does not generates exploits.

Second, researchers have adopted dynamic analysis focusing on real-world transactions to detect vulnerabilities. The aforementioned EasyFlow [12] is one such work that propagates taints to detect integer overflows in runtime. Zhang [46] proposed TXSPECTOR to detect Re-entrancy, UncheckedCall and Suicidal vulnerabilities. GROSSMAN [47] defines a safety property, called ECF, to prevent the exploitation of the vulnerability in the DAO. SODA [48] is an online detection framework supporting verification of inconsistencies between a contract and a corresponding standard or user-defined pattern. Rodler [49] proposed a tool called Sereum to protect existing contracts from reentrancy attacks. Their technology is based on runtime monitoring and verification. Sereum effectively protects the security of the contracts which are already published. Livshits [50] proposed a new DoS attack aimed at exploiting EVM by generating resource exhaustive contracts, whose throughputs are significantly slower than typical contracts. As a comparison, these dynamic analysis works depend on the existence of real-world exploit transactions—EXGEN can generate exploits even if there are no existing transactions.

Third, people have also proposed to formally verify smart contracts or patch them. Bhargavan et al. [51] translates Solidity into F*, a functional programming language designed for better formal verification. Grishchenko et al. [52] further formalizes EVM bytecode in the F* proof assistant, and proposes eThor [53] for automated and static analysis of EVM bytecode. Tikhomirov [54] translated Solidity into an XML-based IR, and checked if the contracts violate the $XPath$ patterns. EthBMC [55] adopts bounded model checking to detect vulnerabilities. EVMPatch [56] has proposed to automatically update and patch vulnerable contracts via a proxy pattern. None of these works can automatically generate exploit for smart contracts.

Blockchain Security and Privacy. We describe some general works on blockchain security and privacy in this subsection. Many prior works [24], [57]–[62] illustrated the basic framework of Ethereum and surveyed the security issues of Ethereum contracts from different perspectives. Zhou et al. [11] performed the first comprehensive study of Ethereum transactions and measured real-world adoptions of attacks and defenses. Lee et al. [63] addressed the void in the investigation into EOS blockchain and introduced four attacks (not targeting at integer overflows) whose root causes stem from the unique characteristics of EOS. Delmolino et al. [64] provided their own experience in building safer smart contracts. Konoth et al. [65] proposes MineSweeper to detect cryptojacking based on the intrinsic characteristics of cryptomining code. People also propose to use blockchain together with Tor to protect privacy and security [66], [67]. Other papers [68]–[70] presented their mechanism to motivate security research on contracts.

9 CONCLUSION

In this paper, we propose the first cross-platform, open-source framework, called EXGEN, to automatically generate exploits for smart contracts on EOS and Ethereum. First, EXGEN generates partially-ordered transactional set (PTS) and symbolic attack contracts following data dependencies among different transactions preparing for the vulnerable states. Then, EXGEN symbolically executes the generated contracts to concretize symbolic values by extracting and solving constraints. Our evaluation shows that EXGEN successfully outperforms state-of-the-art tool, namely teEther, in generating exploits for vulnerable Ethereum (Solidity) contracts. The evaluation also shows that EXGEN is able to find 50 zero-day vulnerabilities of 24 EOS smart contracts.

REFERENCES

[1] "SWC-107, Reentrancy," https://swcregistry.io/docs/SWC-107.

[2] "SWC-101, Integer Overflow and Underflow," https://swcregistry.io/docs/SWC-101.

[3] "EOS Smart Contract Security Best Practices," https://github.com/slowmist/eos-smart-contract-security-best-practices/blob/master/README_EN.md#the-real-case, June 2019.

[4] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. ACM, 2016, pp. 254–269.

[5] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts." in *NDSS*, 2018.

[6] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, 2018, pp. 653–663.

[7] B. Mueller, "Smashing ethereum smart contracts for fun and real profit," *HITB SECCONF Amsterdam*, 2018.

[8] C. F. Torres, J. Schütte, and R. State, "Osiris: Hunting for integer bugs in ethereum smart contracts," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 664–676.

[9] E. Lai and W. Luo, "Static analysis of integer overflow of smart contracts in ethereum," in *Proceedings of the 2020 4th International Conference on Cryptography, Security and Privacy*, 2020, pp. 110–115.

[10] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, "Vandal: A scalable security analysis framework for smart contracts," *arXiv preprint arXiv:1809.03981*, 2018.

[11] S. Zhou, Z. Yang, J. Xiang, Y. Cao, Z. Yang, and Y. Zhang, "An ever-evolving game: Evaluation of real-world attacks and defenses in ethereum ecosystem," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2020, pp. 2793–2810.

[12] J. Gao, H. Liu, C. Liu, Q. Li, Z. Guan, and Z. Chen, "Easyflow: Keep ethereum away from overflow," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2019, pp. 23–26.

[13] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi, "Town crier: An authenticated data feed for smart contracts," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. ACM, 2016, pp. 270–282.

[14] B. Jiang, Y. Liu, and W. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 259–269.

[15] J. Krupp and C. Rossow, "teether: Gnawing at ethereum to automatically exploit smart contracts," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 1317–1333.

[16] E. Elrom, "Eos. io wallets and smart contracts," in *The Blockchain Developer*. Springer, 2019, pp. 213–256.

[17] "wasm2c: Convert wasm files to C source and header," https://github.com/WebAssembly/wabt/tree/master/wasm2c.

[18] Google drive link of our code and dataset. https://drive.google.com/file/d/10unHPpARh9FBrVIyarkiSVMKSd3qfGV5/view?usp=sharing.

[19] "llvmlite User Guide," https://llvmlite.readthedocs.io/en/latest/user-guide/index.html.

[20] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[21] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[22] "Ethereum Smart Contract Bug Injection Framework," https://github.com/xf97/HuangGai.

[23] J. Feist, G. Grieco, and A. Groce, "Slither: a static analysis framework for smart contracts," in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 2019, pp. 8–15.

[24] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 67–82.

[25] "The code of Solidity contract "Ethsplit" from Etherscan," https://etherscan.io/address/0xc5b2508e878af367ba4957bdbeb2bbc6da5bb349#code.

[26] "The account information of EOS contract "Gameworldcom" from bloks.io," https://www.bloks.io/account/gameworldcom.

[27] "The code of EOS contract "Blaster" from GitHub," https://github.com/michaeljyeates/eos-blaster.

[28] "The code of Solidity contract "HmcDistributor" from Etherscan," https://etherscan.io/address/0x7329e152d805dec79c67e45d4e04f3b6ce545b7c#code.

[29] "The code of Solidity contract "NumbersToken2" from Etherscan," https://etherscan.io/address/0xd8be633339d08eca913d7d8a05806ef37a896ef2#code.

[30] "The code of Solidity contract "Agricoin" from Etherscan," https://etherscan.io/address/0xf331f7887d31714dce936d9a9846e6afbe82e0a0#code.

[31] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley, "Automatic exploit generation," *Communications of the ACM*, vol. 57, no. 2, pp. 74–84, 2014.

[32] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 380–394.

[33] S.-K. Huang, M.-H. Huang, P.-Y. Huang, H.-L. Lu, and C.-W. Lai, "Software crash analysis for automatic exploit generation on binary programs," *IEEE Transactions on Reliability*, vol. 63, no. 1, pp. 270–289, 2014.

[34] S. Heelan, T. Melham, and D. Kroening, "Gollum: Modular and greybox exploit generation for heap overflows in interpreters," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1689–1706.

[35] W. Wu, Y. Chen, J. Xu, X. Xing, X. Gong, and W. Zou, "{FUZE}: Towards facilitating exploit generation for kernel use-after-free vulnerabilities," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 781–797.

[36] W. Chen, X. Zou, G. Li, and Z. Qian, "{KOOBE}: Towards facilitating exploit generation of kernel out-of-bounds write vulnerabilities," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 1093–1110.

[37] M. Zalewski, "American Fuzzy Lop," http://lcamtuf.coredump.cx/afl/.

[38] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "sfuzz: An efficient adaptive fuzzer for solidity smart contracts," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 778–788.

[39] J. Choi, G. Grieco, D. Kim, A. Groce, S. Kim, and S. K. Cha, "Smartian: Enhancing smart contract fuzzing with static and dynamic data-flow analyses," in *The 36th IEEE/ACM International Conference on Automated Software Engineering*. IEEE/ACM, 2021.

[40] "Mythril Github Project," https://github.com/ConsenSys/mythril.

[41] C. F. Torres, M. Steichen *et al.*, "The art of the scam: Demystifying honeypots in ethereum smart contracts," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1591–1607.

[42] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2017, pp. 442–446.

[43] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "Madmax: Surviving out-of-gas conditions in ethereum smart contracts," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–27, 2018.

[44] Y. Zhou, D. Kumar, S. Bakshi, J. Mason, A. Miller, and M. Bailey, "Erays: reverse engineering ethereum's opaque smart contracts," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 1371–1385.

[45] N. He, R. Zhang, H. Wang, L. Wu, X. Luo, Y. Guo, T. Yu, and X. Jiang, "EOSAFE: Security analysis of EOSIO smart contracts," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/he-ningyu

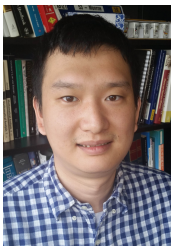
[46] M. Zhang, X. Zhang, Y. Zhang, and Z. Lin, "Txspector: Uncovering attacks in ethereum from transactions," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[47] S. Grossman, I. Abraham, G. Golan-Gueta, Y. Michalevsky, N. Rinetzky, M. Sagiv, and Y. Zohar, "Online detection of effectively callback free objects with applications to smart contracts," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, pp. 1–28, 2017.

[48] T. Chen, R. Cao, T. Li, X. Luo, G. Gu, Y. Zhang, Z. Liao, H. Zhu, G. Chen, Z. He *et al.*, "Soda: A generic online detection frame-

work for smart contracts," in *Proceedings of the 27th Network and Distributed System Security Symposium*, 2020.
[49] M. Rodler, W. Li, G. O. Karame, and L. Davi, "Sereum: Protecting existing smart contracts against re-entrancy attacks," in *NDSS*, 2019.
[50] D. Perez and B. Livshits, "Broken metre: Attacking resource metering in evm," in *Proceedings of the 27th Network and Distributed System Security Symposium*, 2020.
[51] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy *et al.*, "Formal verification of smart contracts: Short paper," in *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*. ACM, 2016, pp. 91–96.
[52] I. Grishchenko, M. Maffei, and C. Schneidewind, "A semantic framework for the security analysis of ethereum smart contracts," in *International Conference on Principles of Security and Trust*. Springer, 2018, pp. 243–269.
[53] C. Schneidewind, I. Grishchenko, M. Scherer, and M. Maffei, "ethor: Practical and provably sound static analysis of ethereum smart contracts," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 621–640.
[54] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 2018, pp. 9–16.
[55] J. Frank, C. Aschermann, and T. Holz, "ETHBMC: A bounded model checker for smart contracts," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020.
[56] M. Rodler, W. Li, G. O. Karame, and L. Davi, "Evmpatch: Timely and automated patching of ethereum smart contracts," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/rodler
[57] S. Wang, Y. Yuan, X. Wang, J. Li, R. Qin, and F.-Y. Wang, "An overview of smart contract: architecture, applications, and future trends," in *2018 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2018, pp. 108–113.
[58] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in *International Conference on Principles of Security and Trust*. Springer, 2017, pp. 164–186.
[59] G. Destefanis, M. Marchesi, M. Ortu, R. Tonelli, A. Bracciali, and R. Hierons, "Smart contracts vulnerabilities: a call for blockchain software engineering?" in *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, 2018, pp. 19–25.
[60] A. Mense and M. Flatscher, "Security vulnerabilities in ethereum smart contracts," in *Proceedings of the 20th International Conference on Information Integration and Web-based Applications & Services*. ACM, 2018, pp. 375–380.
[61] A. Juels, A. Kosba, and E. Shi, "The ring of gyges: Investigating the future of criminal smart contracts," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 283–295.
[62] D. Perez and B. Livshits, "Smart contract vulnerabilities: Does anyone care?" *arXiv preprint arXiv:1902.06710*, 2019.
[63] S. Lee, D. Kim, D. Kim, S. Son, and Y. Kim, "Who spent my EOS? on the (in) security of resource management of eos. io," in *13th USENIX Workshop on Offensive Technologies (WOOT 19)*, 2019.
[64] K. Delmolino, M. Arnett, A. Kosba, A. Miller, and E. Shi, "Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab," in *International conference on financial cryptography and data security*. Springer, 2016, pp. 79–94.
[65] R. K. Konoth, E. Vineti, V. Moonsamy, M. Lindorfer, C. Kruegel, H. Bos, and G. Vigna, "Minesweeper: An in-depth look into drive-by cryptocurrency mining and its defense," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1714–1730.
[66] S. Petrov, S. Kendzierskyj, and H. Jahankhani, "Protecting privacy and security using tor and blockchain and de-anonymization risks," in *Cyber Defence in the Age of AI, Smart Societies and Augmented Humanity*. Springer, 2020, pp. 199–232.
[67] M. A. Rahman, M. S. Hossain, G. Loukas, E. Hassanain, S. S. Rahman, M. F. Alhamid, and M. Guizani, "Blockchain-based mobile edge computing framework for secure therapy applications," *IEEE Access*, vol. 6, pp. 72 469–72 478, 2018.
[68] L. Breindenbach, P. Daian, F. Tramèr, and A. Juels, "Enter the hydra: Towards principled bug bounties and exploit-resistant smart contracts," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 1335–1352.
[69] M. Bartoletti and R. Zunino, "Bitml: a calculus for bitcoin smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 83–100.
[70] H. Kalodner, S. Goldfeder, X. Chen, S. M. Weinberg, and E. W. Felten, "Arbitrum: Scalable, private smart contracts," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 1353–1370.

Ling Jin received her B.Eng. degree in Computer Science and Technology at Xidian University, Xi'an, Shaanxi, China, in 2016. She is currently pursuing the Ph.D. degree with the College of Computer Science and Technology, Zhejiang University, Hangzhou, Zhejiang, China. Her research interests include program analysis, mobile security and blockchain security.



Yinzhi Cao is an assistant professor in Computer Science at Johns Hopkins University. He earned his Ph.D. in Computer Science at Northwestern University and worked at Columbia University as a postdoc. Before that, he obtained his B.E. degree in Electronics Engineering at Tsinghua University in China. His research mainly focuses on the security and privacy of the Web, smartphones, and machine learning. His past work was widely featured by over 30 media outlets, such as NSF Science Now (Episode 38), CCTV News, IEEE Spectrum, Yahoo! News, and ScienceDaily. He received two best paper awards at SOSP17 and IEEE CNS15 respectively. He is a recipient of the Amazon ARA award 2017 and NSF CAREER Award 2021.

Yan Chen received the Ph.D. degree in computer science from the University of California at Berkeley, USA, in 2003. He is currently a professor with the Department of Electrical Engineering and Computer Science, Northwestern University, USA and a distinguished professor with the College of Computer Science and Technology, Zhejiang University, China. Based on Google Scholar, his papers have been cited over 10,000 times and his h-index is 49. His research interests include network security, measurement, and diagnosis for large-scale networks and distributed systems. He received the Department of Energy Early CAREER Award in 2005, the Department of Defense Young Investigator Award in 2007, the Best Paper nomination in ACM SIGCOMM 2010, and the Most Influential Paper Award in ASPLOS 2018.

Di Zhang received his bachelor's degree in Information Security from the College of Computer Science and Technology, Zhejiang University, Hangzhou, Zhejiang, China. His research interests include blockchain security and virtual machines.

Simone Campanoni is a tenure-track assistant professor at the Computer Science department of Northwestern University. He received his Ph.D. in Information Technologies at Politecnico di Milano. Before that, he obtained his M.S. and B.S. degree in Computer Engineering at Politecnico di Milano. His main research area is compilers, with special interest in its relation with computer architecture, runtime systems, operating systems, and programming languages.