Pluto: Exposing Vulnerabilities in Inter-Contract Scenarios

Fuchen Ma, Zhenyang Xu, Meng Ren, Zijing Yin, Yuanliang Chen, Lei Qiao, Bin Gu*, Huizhong Li, Yu Jiang* Jiaguang Sun

Abstract—Attacks on smart contracts have caused considerable losses to digital assets. Many techniques based on symbolic execution, fuzzing, and static analysis are used to detect contract vulnerabilities. Most of the current analyzers only consider vulnerability detection intra-contract scenarios. However, Ethereum contracts usually interact with others by calling their functions. A bug hidden in a path that depends on information from external contract calls is defined as an inter-contract vulnerability. Failure to deal with this kind of bug can result in potential false negatives and false positives. In this work, we propose Pluto, which supports vulnerability detection in inter-contract scenarios. It first builds an Inter-contract Control Flow Graph (ICFG) to extract semantic information among contract calls. Afterward, it symbolically explores the ICFG and deduces Inter-Contract Path Constraints (ICPC) to check the reachability of execution paths more accurately. Finally, Pluto detects whether there is a vulnerability based on some predefined rules. For evaluation, we compare Pluto with five state-of-the-art tools, including Oyente, Mythril, Securify, ILF, and Clairvoyance on a labeled benchmark and 39,443 real-world Ethereum smart contracts. The result shows that other tools can only detect 10% of the inter-contract vulnerabilities, while Pluto can detect 80% of them on the labeled dataset. Beyond that, Pluto has detected 451 confirmed vulnerabilities on real-world contracts, including 36 vulnerabilities in inter-contract scenarios. Two bugs have been assigned with unique CVE identifiers by the US National Vulnerability Database (NVD). On average, Pluto costs 16.9 seconds to analyze a contract, which is as fast as the state-of-the-art tools.

Index Terms—Smart Contracts, Ethereum, Inter-Contract Vulnerabilities

1 Introduction

Smart contracts have attracted the attention of many attackers because they are often associated with money. In recent years, the vulnerabilities in smart contracts have caused huge losses [1], [2], [3], [4]. In Ethereum [5], a contract cannot be modified once it is deployed. So, it is vital to check whether a smart contract has vulnerabilities before deployment. To achieve this goal, researchers use techniques such as symbolic execution [6], [7], fuzzing [8], [9], [10] and static analysis [11], [12] on smart contract testing.

All these techniques have achieved great success in finding lots of real vulnerabilities in smart contracts. However, most of them only consider the detection in intra-contract scenarios without calling another contract. When encountering inter-contract calls, they always fail to make a thorough analysis. Nevertheless, contracts on Ethereum call functions from another contract. This leads to lots of false negatives and false positives by existing detection tools. Different from inter-procedural analysis for traditional programs, analysis on smart contracts need to deal with more types of call operations (CALL,CALLCODE,STATICCALL and DELEGATECALL). Besides, the execution of the caller contract needs to initialize a new stack in Ethereum. For these reasons, the techniques used in the traditional inter-procedural analysis cannot be used directly in smart contracts. Detecting vulnerabilities in inter-contract scenarios struggles with two significant challenges.

The first challenge arises in collecting semantic information from inter-contract scenarios. Critical semantic information, such as money flow, is essential for smart contract vulnerability detection. Money flow means how the tokens transfer among accounts during the contract execution [13]. However, due to the lack of inter-contract state information maintenance, traditional tools cannot correctly track the semantic information under inter-contract scenarios. The second challenge arises in checking path reachability during contract calls. The reachability depends on whether certain variables can be calculated correctly in inter-contract scenarios. The parameters of a contract call may depend on the context of the caller contract. Meanwhile, the results of the contract call may affect the following execution paths. There is no proper way to derive the correct scope of these variables during the transmission among contracts, which leads to false judgments on path reachability.

To resolve the above challenges, we propose Pluto, which aims at detecting vulnerabilities under inter-contract scenarios. First, Pluto constructs an Inter-contract Control Flow Graph (ICFG) to collect the semantic information from inter-contract scenarios correctly. An ICFG contains blocks from the caller contract's CFG, and the blocks corresponded with the called function from the callee contract's CFG. Then, Pluto explores the ICFG and deduces Inter-Contract Path Constraints (ICPC) for the call operation. In this way, Pluto can derive the correct scope of variables during inter-contract calls and adequately identify whether a path is reachable by using the SMT solver tool such as Z3 [14]. Finally, Pluto will validate whether a vulnerability exists according to predefined rules for each reachable path. For now, Pluto supports the detection of the three types of vulnerabilities: integer overflow, timestamp dependency, and reentrancy. They account for 97.64% of CVEs assigned to Ethereum smart contracts according to statistics [15].

For evaluation, we compared Pluto with five state-of-art tools including Oyente [6], Mythril [16], Securify [11], ILF [10] and Clairvoyance [17]. We evaluate Pluto on three datasets. The first one contains 150 contracts with 150 manually injected inter-contract vulnerabilities. The second one is from SolidiFI dataset [18] which consists of 898 injected intra-contract vulnerabilities. The third one is the dataset from SmartBugs [19] which includes 39,443 real-world contracts deployed on Ethereum. The result on the first dataset demonstrates that Pluto can report 80% of the inter-contract vulnerabilities, while other tools can only report 10% oc-

*Bin Gu and Yu Jiang are corresponding authors.

casionally on the labeled benchmark. Meanwhile, on the second dataset, Pluto is proved to perform better than other tools on intra-contract vulnerabilities detection as well. Specifically, Pluto can report 2.33%-70.97% less false positives and 2.48%-90.08% less false negatives than other tools on intra-contract bugs detection. The reason for this result is that Pluto uses a more accurate bug validation strategy than other tools, we will describe this in detail in Section 4. As for real-world smart contracts, Pluto has detected 451 valid vulnerabilities. Among them, 36 vulnerabilities are hidden in inter-contract scenarios. In terms of time, Pluto takes an average of 16.9 seconds to analyze a contract, which is 1.1 seconds slower than Oyente and 15.0-214.9 seconds faster than other tools.

To summarize, our contributions are as follows:

- To detect vulnerabilities in inter-contract scenarios, we propose the approach of inter-contract call analysis. This approach can simulate the call logic among contracts and gain a global perspective for contract semantics.
- We implement and open-source Pluto[1], a vulnerability detection tool that supports inter-contract scenarios. It constructs an ICFG to track the semantic information and deduces ICPC to check path reachability properly. Pluto uses a vulnerability validator to check whether the current contract path is vulnerable.
- We apply Pluto on 39,443 real contracts. It has confirmed 451 previously unknown vulnerabilities, 36 of which are related to inter-contract scenarios. Two bugs are assigned with CVE identifiers by US National Vulnerability Database(CVE-2020-24837 and CVE-2020-24838.).

The rest of the paper is organized as follows: Section 2 describes some necessary background. Section 3 introduces an example of vulnerabilities hidden in inter-contract scenarios and explains the limitations of current tools. In Section 4, we describe the technical detail of Pluto. We illustrate how we implement and evaluate Pluto in Section 5. In Section 6, we discuss some limitations of Pluto. The following section introduces some work related to ours. In the last section, we make a summary of the paper.

2 Background

In this section, we will give a brief introduction to Ethereum smart contracts execution and inter-contract scenarios.

2.1 Ethereum Contracts Execution

In Ethereum, smart contracts are written in a high-level language named Solidity. In order to execute the code and give out the same result in each node, Ethereum develops a virtual machine called EVM. EVM can translate the bytecode of the contract into a sequence of opcodes. Users in Ethereum can commit a transaction that calls a function in a smart contract. A series of transactions will be packaged to a miner node and be executed by the EVM. When executing a transaction, the EVM uses a stack to record the following opcodes and operands. In order to store data, EVM uses either memory or storage. Data stored in the memory is transient, while data in the storage is persistent.

1. https://github.com/PlutoAnalyzer/pluto/tree/main/PlutoTool

To symbolically execute the Ethereum smart contracts, one needs to imitate the execution model of EVM. To be specific, one needs to construct a stack used to record the opcodes and their operands just like EVM does. The difference is that the elements in the stack may be a symbol rather than a concrete value. Apart from that, a symbolic execution tool also needs to define the behavior of each opcode as same as Ethereum. As for the memory and storage, the tool also needs to construct a model which acts the same as the EVM. Then, the tool needs to decide the execution sequence of the smart contracts. Generally, existing tools execute the functions in a contact one by one.

For example, to symbolically execute the example contract listed in Fig 2, a tool needs to initialize a stack, a memory model, and a storage model first. Then, it will try to call the function 'invest' in the example contract at line 22. All the parameters are passed in with symbols. After that, the tool will execute each statement and collect constraints when a branch is encountered. In our example, the tool will first collect 'rule == 0xa214bd6...' as shown in line 26 in order to execute the following statements. For each new path, a symbolic execution will try to solve the constraints along the path to see whether the path is reachable. For example, to trigger the bug at line 31, the tool needs to check whether such constraints can be solved: '(rule == 0xa214bd6...) AND (rule.getThreshold(target) == 203000) AND (203000 ¡= investToken)'. Symbolic tools often use a search algorithm such as Depth First Search to look for all the possible paths in a contract. When all the paths are found, an execution will finish.

2.2 Inter-Contract Scenarios in Ethereum

Ethereum is considered as the second large blockchain system in the world, right after the Bitcoin system. Ethereum smart contracts are programs that can be executed by user accounts in Ethereum. Such an execution process is called a transaction. Ethereum can be viewed as a state machine: $s := c \xrightarrow{t} \bar{c}$, where each state migration is triggered by a series of transactions. Ethereum smart contracts are usually written in a Turing complete language named Solidity. Each node has a virtual machine named Ethereum Virtual Machine(EVM) to parse the Solidity code into a sequence of opcodes and execute the opcodes under a runtime stack.

On Ethereum, calling another contract's function is common in a transaction. A contract can call another contract deployed on Ethereum by referring to its address. This process is implemented by four opcodes: 'CALL', 'DELEGATECALL', 'CALLCODE' and 'STATICCALL'. The differences among them are shown in Fig. 1.

The 'CALL' opcode calls a method in another contract with the storage of the called contract. Besides, a 'CALL' opcode is also used to transfer tokens in Ethereum. An account in Ethereum can perform transfer operations by calling $send()$ or $transfer()$ functions. Both of them invoke the 'CALL' opcode. They are all part of the API in Solidity language. Both functions are underlying calls to the 'CALL' opcode.'DELEGATECALL' opcode calls a method in another contract using the storage of the caller contract. The execution context will not change under 'DELEGATECALL' opcode. That means the called contract can modify the caller contract's storage in some cases. For example, as Fig 1 shows, when contract B calls contract C with 'DELEGATECALL', C modifies the variable 'x' in contract

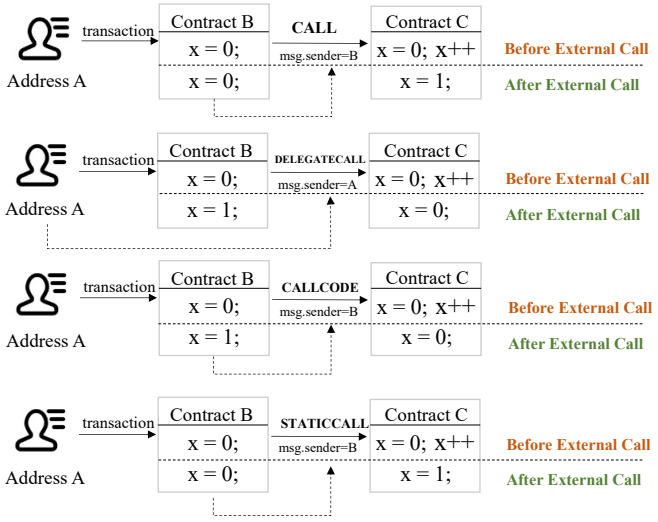


Fig. 1: Four opcodes to call functions in another contract. The part below the dotted line indicates the results after the contract call. 'DELEGATECALL' regards the caller as the sender of the transaction. While 'CALL' and 'CALLCODE' set the sender as the last caller. The execution context of 'CALL' is in the callee contract, while the context of the other two opcodes is in the caller contract. 'STATICCALL' behaves equivalently to 'CALL' except it limits any operation that modify the state.

B. However, with 'CALL' opcode, C modifies 'x' in its own scope. So, 'DELEGATECALL' is considered dangerous and is not recommended unless the delegated address is trusted.

'CALLCODE' is a previous version of 'DELEGATECALL' which updates the sender of the transaction at each call operation. As Fig 1 shows, if B calls contract C with 'CALL' or 'CALLCODE', the sender of the transaction is B. If B calls C with 'DELEGATECALL', the sender is A. Ethereum has deprecated 'CALLCODE' in new versions of Solidity and EVM. The detailed information of these opcodes can be found in Ethereum yellow paper [20]. The opcode 'STATICCALL' behaves equivalently to the opcode 'CALL', except it limits all the operations that change the current state. The limited opcodes contain: 'CREATE','SSTORE', 'LOG0' etc.

3 Overview

We present an example to illustrate the vulnerabilities in inter-contract scenarios. Then we discuss the limitations of existing tools and explain why they fail to detect these vulnerabilities. At last, we demonstrate how Pluto works on the example.

3.1 Motivating Example

Fig 2 shows two contracts. The first contract, 'RuleContract', defines the threshold calculation rules of investment. The threshold depends on the target year of investment. As the function 'getThreshold' shows, the threshold of the year 2020 is 50,000, and the threshold of other targets is equal to 100 multiplied by the target year.

The other contract, 'InvestContract', is used to invest tokens to each target. The contract defines a function named 'invest'. This function takes in three parameters: an address

```
1  contract RuleContract{
2    uint private base_threshold = 50000;
3    function getThreshold(uint256 currentTarget)
4              public returns(uint256){
5      require(currentTarget >= 2020 &&
           currentTarget <= 2050);
6      if(currentTarget == 2020){
7      // result is 50000;
8        return base_threshold;
9      }
10     else{
11     // the max result is 205000;
12       return currentTarget*100;
13     }
14   }
15 }
16
17 contract InvestContract{
18   uint public total_2030 = 0;
19   uint public cur_invest = 0;
20   // some fixed address for rule contract
21   address fixed_rule = "0xa214bd6......"
22   function invest(
23     RuleContract rule,
24     uint target,
25     uint investToken) public {
26     assert(rule==fixed_rule);
27     // the max threshold is 205000;
28     uint threshold = rule.getThreshold(target);
29     if(threshold == 203000 && threshold <=
           investToken){
30       // overflow occurs;
31       total_2030 += investToken;
32       cur_invest = investToken;
33     }
34     else if(investToken > 300000){
35       // no underflows in this branch;
36       cur_invest = investToken - threshold /
             100;
37     }
38   }
39 }
```

Fig. 2: Two contracts used to raise money for several targets. The first sets the investment threshold rules. The second provides a method to invest tokens for each target.

of the contract 'rule', an integer representing the investment target, and an integer storing the number of tokens for investment. This contract also defines two variables to represent total tokens on target 2030 and current invest tokens. As line 36 shows, if the investment amount is more than 300,000, 1% of the threshold value will be deducted as the handling charge.

Vulnerabilities. Contract 'InvestContract' has an overflow vulnerability at line 31. If a malicious investor inputs a large value of 'investToken,' there will be an overflow in the add operation. Attackers need at least two transactions to exploit this vulnerability. Each transaction calls the function 'invest' whose inputs contain: 1) a valid address where the 'RuleContract' is deployed; 2) 2030 as the invest target; 3) a really large value for 'investToken' (input 2^{256}-1 for example). The attackers can successfully set the variable 'total_2030' to a small value though many big investments have been made.

This overflow bug requires interprocedural analysis because the bug is hidden under a condition statement which requests the result of an external call. The condition of the branch is 'threshold == 203000 threshold ¡= investToken'. Though there are no constraints on investToken's value, the

value of the threshold is fetched after the external call. This blocks the analysis tools from finding this bug unless the analysis is inter-procedural.

Although the sub operation at line 36 does not check whether underflow occurs, we will not consider it as a vulnerability. According to the threshold rules defined in 'RuleContract', the value of 'investToken' will always be bigger than the value of 'threshold'. So no one can make an attack through this operation.

3.2 Challenges in Inter-Contract Scenarios

We tested the example with several state-of-art works, including symbolic execution tools like Oyente [6] and Mythril [16], fuzzing tools like sFuzz [9] and ILF [10], and static analysis tools like SmartCheck [21] and Securify [11].

TABLE 1: The results of existing tools on the example. FN refers to the false negative at line 31 in Fig 2. FP refers to false positive at line 36. The grey circle indicates such a problem, while the white one means no such problem.

Tools	Oyente	Mythril	sFuzz	ILF	SmartCheck	Securify
FN	●	●	●	●	●	●
FP	●	●	○	○	○	○

As Table 1 shows, all the tools[2] fail to report the true vulnerability at line 31. Besides, both symbolic execution tools report the add operation at line 36 as a vulnerability which is actually a false alarm. Generally, the existing tools fail in inter-contract scenarios for two reasons: 1) They fail to collect semantic information from inter-contract logic; 2) They fail to check the path reachability correctly.

Semantic information from inter-contract logic. The definition for 'semantic information' is the information flowing between the caller and callee contracts which are needed for the bug detection process. The semantic information is quite important for smart contract analysis. For example, the detection of a timestamp dependency vulnerability needs the information of the money flow. In this example, the variable 'threshold' is set by the logic from contract 'RuleContract'. However, existing symbolic execution tools and static analysis tools construct their analysis only based on the contract 'InvestContract'. The semantic information of 'threshold' cannot be appropriately fetched.

Currently, existing symbolic execution tools always define vulnerability detection rules based on semantic information from a single contract CFG. In inter-contract scenarios, they always fail to combine semantic information from caller and callee contracts. Similarly, static analysis tools such as Securify leverage intermediate representations and checks the representations against some patterns. However, the construction of intermediate representation is usually based on a single contract. In inter-contract scenarios, these representations are incomplete for semantic analysis.

Path reachability during contract calls. In inter-contract scenarios, path reachability depends on the value scope of contract variables in both caller and callee contracts. The inputs of a function call depend on the calling contract, while the results of a function call depending on the called contract. For example, variable 'threshold' in our example in Fig 2 represents the result of the function 'invest'. From the view of contract 'InvestContract', it is hard to know the constraints of 'threshold'. Similarly, it is hard to get the constraints of 'currentTarget' only within the scope of 'RuleContract'. In order to properly tackle the contract call scenarios, we need a global perspective to collect the constraints for inputs and results of calling functions. In this case, the constraint of variable 'threshold' from a global view is:

$$Or(threshold = 50000, (202000 \leq threshold \leq 205000))$$

With this information, we can find out that the overflow at line 31 can really happen with a large value of 'investToken', while the underflow at line 36 can never happen. However, the existing symbolic execution and fuzzing work struggle in generating the correct value scope of 'threshold'.

We use Oyente to illustrate the limitations in symbolic execution tools. As defined by Ethereum, a CALL opcode takes 7 elements from the stack. The first element represents the gas limit for this CALL. The second one sets the address of the caller. The third one specifies the value of Ether passed to the CALL. The following four elements represent the memory offset and size of the input and output, respectively. The result of the CALL opcode will be stored into the stack where 1 represents success and 0 represents failure.

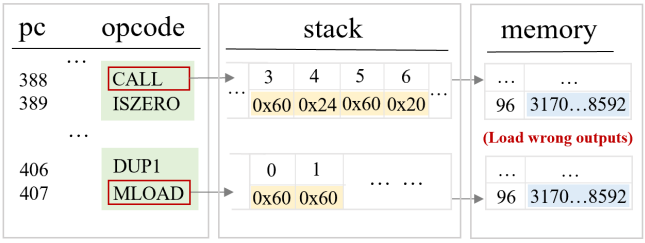


Fig. 3: The state of the stack and memory during the symbolic execution process of Oyente for the example. Loading call outputs incorrectly misleads the subsequent symbolic execution process.

As Fig 3 shows, the offset of the call data and the one of the return data are both '0x60' in our case. This means the inputs of the call store are at the same place as the outputs. The size of the inputs is 36(0x24) bytes, while the size of outputs is 32(0x20) bytes. The extra 4 bytes are used to store the signature of the called function. In this case, the signature can be calculated with $keccak256('getThreshold(uint256)')$, and the result is '0x4615d5e9'.

When processing with the 'CALL' opcode, Oyente first pops seven elements from the stack and then pushes back '1' to the stack which indicates that the CALL is successful. However, when the opcode 'MLOAD' tries to load the return data from memory, it gets '0x4615d5e90000...'(56 zeros are omitted for saving space). The decimal representation of this number is the '3170...8592'(69 digits are omitted for saving space) in Fig 3. In our case, this huge number has been assigned to the variable 'threshold' at line 25 in Fig 2. Consequently, the first condition of the 'if' branch at line 26, which is $threshold == 203000$ cannot be satisfied and the 'else' branch will be executed. This leads to the false negative at line 31 and the false positive at line 36 given by Oyente. In summary, the low-level description of Oyente aims to explain why symbolic execution tools can report

2. Though ILF doesn't support the detection of overflow vulnerability, we found it failed to cover both branches in our example.

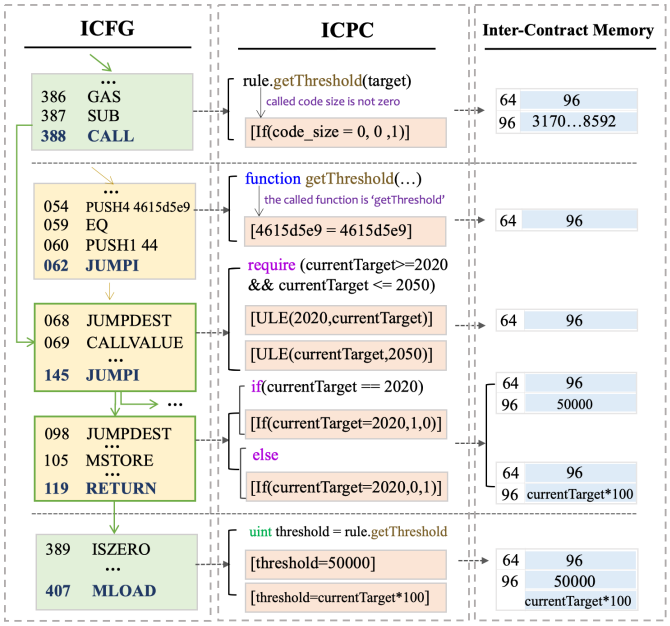


Fig. 4: The workflow of Pluto on the example. Pluto explores the contracts based on ICFG and generates ICPC for path reachability checking. Meanwhile an inter-contract memory is used to store all possible return values in a key-value format.

the false positive and fail to report the true bug. This is not specific to this example; the discussion is suitable for Oyente and Mythril on most inter-contract call scenarios.

In this case, fuzzing tools can hardly generate high quality seeds to check the path reachability correctly. Let us take ILF [10] as an example. It deploys all contracts on a private chain and try to find the correct contract to call. However, it's hard to know that the parameter 'target' should satisfy the constraints $2020 \leq currentTarget \leq 2050$ defined at line 5 in Fig 2. Though ILF uses a symbolic expert to guide the fuzzing process, it fails to generate great seeds in inter-contract scenarios. We used ILF to test the example and found that ILF only achieves 69.75% instructions coverage for contract 'InvestContract'. While, if we delete the precondition of the branch, ILF will cover all the instructions and detect the bug.

3.3 Pluto on the Example

In order to address the challenges and make up for the limitations mentioned above, we designed and implemented Pluto. Pluto explores the contracts on an Inter-contract Control Flow Graph (ICFG), connecting the caller contract with the called one. Pluto will then deduce Inter-Contract Path Constraints (ICPC) while exploring to check the path reachability by deriving the correct constraints for contract variables. Fig 4 describes the workflow of Pluto on the example. We will explain each column in the figure to demonstrate the working principle of Pluto.

Inter-contract Control Flow Graph (ICFG). Pluto combines the caller contract's CFG with the called contract's CFG, and we name the combined graph 'Inter-contract Control Flow Graph'. The details of building an ICFG are described in Section IV. The first column in Fig 4 shows the ICFG on the example. Two green blocks in the figure are taken from the CFG of the caller contract 'InvestContract'. In the original CFG, they belong to the same block. In order to execute the call operation properly, Pluto splits the block and connects blocks in called contract's CFG after 'CALL' opcode.

The yellow blocks in the figure represent the blocks in called contract's CFG. The first block specifies which function is called. In our case, the called function is 'getThreshold', which is represented by its signature in the CFG. The next two yellow blocks represent the execution process of the called function. Pluto will back up the current contract state first. Besides, Pluto pushes some necessary variables into the called contract's storage and memory. After that, Pluto will connect those blocks right after the CALL opcode and execute the called function directly. A RETURN opcode indicates that the called function has finished. Pluto will connect the RETURN blocks with the opcode after CALL in the original contract. With the help of the ICFG, Pluto can execute contracts with inter-contract calls successfully.

Inter-Contract Path Constraints (ICPC). Though ICFG can help Pluto walk through the called functions, it cannot correctly judge path reachability. In order to get a proper constraint for the inputs and outputs of a call operation, we design a series of deduction rules to generate ICPC. Besides, we use an inter-contract memory to store all the possible return values of a call. ICPC is a set of path constraints. However, it is not just the union of the path constraints of the caller and callee contracts. ICPC can correctly handle constraints on variables that flow between the caller and the callee contract. We now describe how Pluto checks the path reachability in this case.

First, before executing a CALL opcode, Pluto will collect all the path constraints and store the call data into the inter-contract memory. In this case, the constraint is $If(code\_size = 0, 0, 1)$ as shown in the first red block in Fig 4. This constraint makes sure that the called code size is not zero. Then Pluto stores the call data of the called function.

When Pluto enters the called contracts, it finds the called function based on the function signature as described in the constraint '$4615d5e9 = 4615d5e9$'. This checks whether the called function is 'getThreshold' and the result is always 'TRUE'. After that, Pluto starts to execute the 'require' statement in function 'getThreshold'. This statement generates two constraints for variable 'currentTarget': $ULE(2020, currentTarget)$ and $ULE(currentTarget, 2050)$. 'ULE' refers to 'Unsigned Less than or Equal to' here. These constraints indicate that the scope of variable $currentTarget$ is $2020 \leq currentTarget \leq 2050$. If the 'require' statement is satisfied, Pluto will enter into the 'if-else' branch of the function. The path constraints for 'if' branch will be added with $If(currentTarget = 2020, 1, 0)$ while the ones for 'else' branch will be added with $If(curr$
$entTarget = 2020, 0, 1)$.

When encountered with a RETURN opcode, Pluto will store all possible return values into the inter-contract memory. In this case, the return value can be '50000' or 'currentTarget * 100'. To distinguish different return value types, we consider '50000' as a real number and 'currentTarget * 100' as an expression. They will both be stored in address '96'(0x60) of inter-contract memory. Pluto then calls back to the original contract and executes the following opcodes.

Then Pluto uses the 'MLOAD' opcode to load the return value of the CALL. If the return variable is a real number, Pluto will add an equation to the current

path constraints like: $threshold = 50000$. Otherwise, if the return value is an expression, Pluto will first add $threshold = currentTarget * 100$ into the path constraints. Then, it collects all the constraints related to the variables in the expression and adds them to the path constraints. In this way, Pluto can know that the function call result is $currentTarget * 100$ or 500000. With the knowledge of the range of $currentTarget$, Pluto can derive the correct scope of the CALL results.

With ICPC, Pluto can accurately check the path reachability in inter-contract scenarios. In this way, it can successfully detect the overflow at line 31 in the example. As for the false positive at line 36, Pluto can successfully eliminate them because it knows that the variable 'threshold' can never be bigger than 205000.

4 Design of Pluto

In this section, we will formally introduce the design of Pluto. Fig 5 shows the overall architecture of Pluto. Generally, the inputs of the system are the target contract together with all of the related contracts, which are called the target contract. The input contracts can be in source or bytecode form. The system's outputs are the contract's vulnerability report and the advice for avoiding such kind of vulnerability for developers. In the Pluto system, there are three general steps for the analysis of a contract.

The first step is to construct an ICFG. In this step, Pluto constructs CFG for the caller contract and the callee contracts. After that, Pluto initializes the ICFG. During the execution process, Pluto supplements the ICFG with dynamic transition information obtained from the runtime stack. In the second step, Pluto collects the path constraints for each contract and deduces ICPC when exploring the ICFG. Pluto will first deduce the ICPC for call inputs. When the call ends, Pluto generates ICPC for call outputs according to the return value form. Finally, Pluto will validate whether each path has a vulnerability with the help of some predefined vulnerability rules. If there is a vulnerability found, Pluto will generate a bug report and advise on how to deal with such a bug. We now introduce each step in detail.

4.1 ICFG Construction

Before making a thorough analysis of smart contracts, Pluto will first setup an ICFG. An ICFG can be described as the definition 1. B denotes the blocks in the target contract's CFG where all blocks containing a call operation have been split. C represents the blocks related to the called function is called contract's CFG.

$$G \equiv \{b_t : b_t \in B, b_d : b_d \in C, \delta\} \quad (1)$$

δ means the transitions among blocks. We will now describe how an ICFG is constructed in detail.

CFG Construction. To build an ICFG, Pluto should build CFG for the target contract and called contracts first. Blocks in a contract CFG contain an opcode sequence ended with opcode 'JUMP' or 'JUMPI'. Edges in a CFG connect the blocks to illustrate the jump target for each jump operation. The main challenge of building a CFG for a contract is to establish the edges properly. Pluto accomplishes this task with two stages.

In the static stage, Pluto will link all the static edges in a CFG first. The static edge is the edge between the block ended with the 'JUMPI' opcode and one of the jump target blocks. 'JUMPI' opcode has two destinations: if the top element of the current stack is zero, the destination is the opcode right after 'JUMPI'. Otherwise, the destination is represented by the second element of the stack. Though the second destination cannot be determined without execution, the first target is specified before execution. Pluto will draw these edges to the CFG in this stage.

Pluto will maintain a runtime stack during the dynamic stage to store the opcodes and operands for the contract. When the opcode 'JUMP' or the opcode 'JUMPI' is encountered, Pluto can read the elements in the stack and get the proper jump destination for each block. Pluto can complete the CFG during this process. According to a previous work, [22], due to the reasons like incomplete code patterns, symbolic execution tools may fail to construct CFG sometimes. This can be enhanced by completing the CFG with the control flow transfers covered by traces. We will enhance Pluto with this technique in the future.

Algorithm 1: Algorithms of building ICFG in Pluto

```
   Input: CFG_t, CFG_c: CFG of the target and the called
          contract.
   Output: ICFG: Inter-contract control flow graph.
 1 Function Main(CFG_t,CFG_c):
 2     split_CALL(CFG_t)
 3     ICFG = CFG_t
 4     while isExecution do
 5         Blocks = getAllBlocks(CFG_t)
 6         for block in Blocks do
 7             if 'CALL' in block then
 8                 edgeICFG(block,CFG_c,ICFG)
 9             end
10         end
11     end
12     return ICFG
13 End Function
14 Function split_CALL(CFG_t):
15     Blocks = getAllBlocks(CFG_t)
16     for block in Blocks do
17         if 'CALL' in block then
18             newblocks = block.split('CALL')
19             Blocks.delete(block)
20             Blocks.insert(newblocks,Blocks.getIndex(block))
21         end
22     end
23 End Function
24 Function edgeICFG(block,CFG_c,ICFG):
25     c_Blocks = getProperBlocks(CFG_c,block)
26     ICFG.connect(block,c_Blocks.first())
27     for c_block in c_Blocks do
28         if 'RETURN' in c_block then
29             connect(c_block,block.next())
30         end
31     end
32 End Function
```

After establishing the CFGs, Pluto can build an ICFG based on them. Algorithm 1 describes the workflow of building an ICFG. As shown in lines 2 and 3, Pluto first splits the 'CALL'('DELEGATECALL', 'CALLCODE', 'STATICCALL') block in the target contract's CFG and initializes the ICFG. As described from line 6 to line 10, Pluto supplements the ICFG during the execution process by adding dynamic edges.

ICFG Initialization. As shown from lines 14 to 23 in Algorithm 1, Pluto splits the target contract's CFG before the ICFG construction. First, Pluto will get all the blocks in the CFG of the target contract, as shown in line 15. For each block. Pluto checks whether there is a 'CALL' opcode (or 'DELEGATECALL', 'CALLCODE', 'STATICCALL'). The

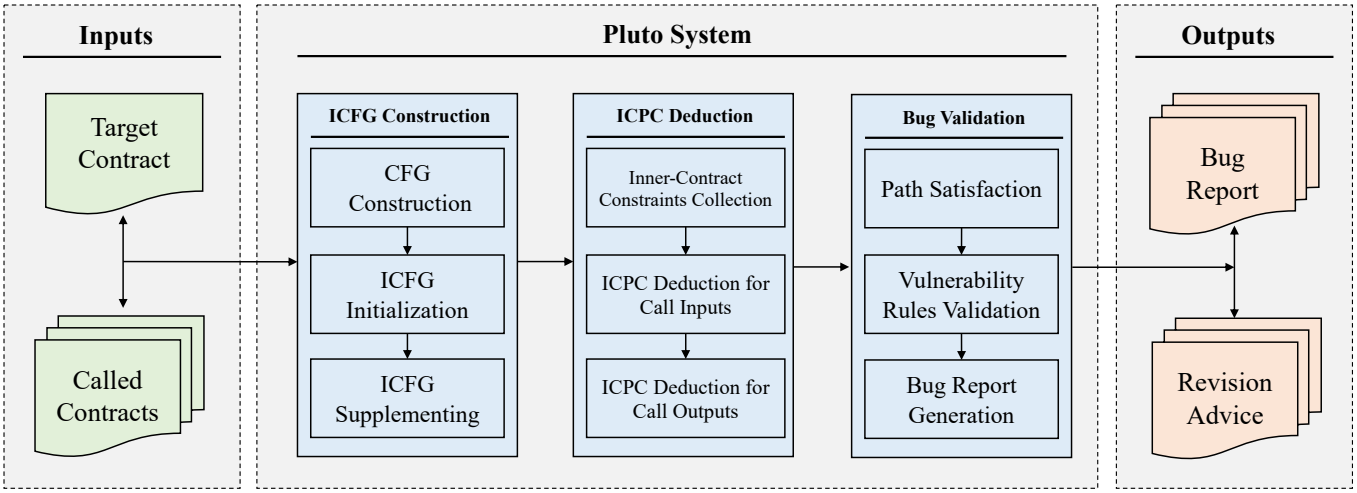


Fig. 5: The overall architecture of Pluto. The inputs of the system are the target contract as well as the called contracts. The outputs are bug reports and some advice for developers on those vulnerabilities. There are three steps in Pluto: ICFG construction, ICPC deduction and bug validation. With these steps, Pluto can successfully support the bug detection in inter-contract scenarios.

block with the call operation will then be split into two blocks, as line 18 shows. The first block contains a call operation and all the opcodes before it. The second block consists of all the opcodes after that call operation. As described in line 19, the original block will be deleted then, and both new blocks will be inserted into the block list. This splitting process aims at preparing for the subsequent splicing process with the called contract. After the initialization, the block set B from definition 1 has been constructed. The definition of B is:

$$B \equiv B_a \backslash \{b_c, b_c \in B_a \wedge CALL \in O[b_c]\} \cup \{b_n, b_n \in B_n\} \quad (2)$$

B_a denotes all the blocks in the target contract and O means the opcodes of certain block. b_c means the blocks in B_a with a call operation. b_n stands for new blocks generated after the splitting of blocks with call operations.

ICFG Supplementation. Line 24 to 32 in algorithm 1 describes how Pluto supplements the ICFG. Pluto will first get the called function for each block in the ICFG that contains the call operation. The first function in the called contract that has the same signature as the called function is considered the proper function. This defines the symbol C in definition 1.

Pluto resolves the call targets by sending a request to the Etherscan, an explorer for Ethereum blockchain. The request contains the address of the callee contract. If the address is a constant one, Pluto can use this approach to resolve the target function. The calling process will not do a query to a node because Pluto has constructed a storage model similar to Ethereum. Any content in the storage can be found in such a model. However, this process may not have success sometimes when the called function's address is passed as the parameter by users. This issue is discussed in Section 6.

$$C \equiv \{p, p \in G[b_s]\} \quad (3)$$

As shown in definition 3, b_s represents the block right after the block which contains the signature of the called function. $G[b_s]$ means the subgraph starts with the block b_s. Pluto will then connect the block which has the call operation with the first block from C. Each block with the 'RETURN' opcode in C should be connected with the block right after the block with the call operation from the original CFG. This defines the symbol δ from definition 1 as:

$$\delta \equiv \delta_t \cup \{(b_c, b_s)\} \cup \{(b_r, b_n), RETURN \in O[b_r]\} \quad (4)$$

δ_t denotes the transitions among blocks in the ICFG and b_r represents the blocks in the called function with 'RETURN' opcode. After the connecting, the current call operation can be successfully spliced with the original ICFG, and Pluto can make a global perspective on both the caller contract and the callee contract.

4.2 Inter-Contract Path Constraints Deduction

After the construction of ICFG, Pluto will explore the graph and generate the Inter-Contract Path Constraints(ICPC) by customized Hoare logic. Before that, Pluto will first initialize some machine states and global variables for intra-contract constraints collection. During the purely symbolic execution of the call, Pluto deduces the ICPC to get the correct value scope of the contract variables.

Intra-Contract Constraints Collection. In order to collect path constraints in a single contract, Pluto needs to define some machine states and global variables to imitate the execution of contracts. We use the symbol μ to denote the machine states which is defined as definition 5: μ_{sta} represents the stack states, μ_{mem} denotes the inter-contract memory and μ_{sto} means the storage states.

$$\mu \equiv \{\mu_{sta}, \mu_{mem}, \mu_{sto}\} \quad (5)$$

The stack is used to record the execution parameters and results of different opcodes. In Pluto, the elements in the stack can be either a real number or a symbolic expression. The inter-contract memory is used to record the inputs and outputs of a call operation. Fig 4 gives an example of the working principle of inter-contract memory. The difference between the inter-contract memory and the actual memory model in Ethereum lies in two aspects: 1) One address in inter-contract memory may be mapped to multiple values.

2) The elements stored in the inter-contract memory can be a symbolic expression. Pluto abstracts the memory slots in the actual memory model in the inter-contract memory model. The actual slots are abstracted as "one-to-many" dictionaries in the inter-contract memory model. This is used to record all the possible results of the external call. This model has no difference from the actual model on the intra-contract execution. On the inter-contract execution, the model only records more information than the actual model. So, the only impact may be that this model will cost more resources. The storage is a persistent memory to store some state variables and some local variables such as mappings and arrays. Pluto will initialize these machine states and use them to simulate the real execution process. Apart from that, Pluto also needs to add some global contract variables into the storage.

Global contract variables refer to the global variables defined in a contract. All the contract variables are stored in the storage. Pluto needs to store the proper values of these variables before the execution to collect path constraints related to the contract variables. The initialization of the contract variables is done during the deployment phase. Pluto uses solc [23], a solidity compiler, to get the opcode sequence of the deployment phase. For each 'SSTORE' opcode, Pluto gets the value and location to be stored in the storage.

After this process, Pluto will then execute each opcode under the rules defined by Ethereum. Each opcode in Ethereum can be defined as symbol σ in definition 6.

$$\{\hat{\mu_{sta}}, \hat{\mu_{mem}}, \hat{\mu_{sto}}\} \equiv \sigma(\{\mu_{sta}, \mu_{mem}, \mu_{sto}\}) \tag{6}$$

The symbol with a hat refers to the new machine state after the execution of the opcode. That means the execution of an opcode leads to a transition from an old machine state to a new one. During the execution, Pluto collects the path constraints according to the elements in the machine states until a call operation is encountered. In order to help understanding, we will follow the motivating example in the rest of the description.

ICPC deduction for Call Inputs. When a call operation is encountered, Pluto needs to execute the called function with proper inputs. First, Pluto uses a global state recorder to backup the current state. This process can be defined by the customized Hoare logic rule:

$$\frac{\{\mu\}\ \mu := rec(\mu_{sta}, \mu_{sto})\ \{\hat{\mu}\}, \{\hat{\mu}\}\ \hat{\mu} := init()\ \{\mu_n\}}{\{\mu\}\ \mu := backup(\mu)\ \{\mu_n\}} \tag{7}$$

The symbol μ represents the machine states, while $\hat{\mu}$ and μ_n represents the temporary states after recording and the new states after the backup process. Pluto only stores the stack and storage states when calling another contract. The inter-contract memory will not be initialized in the new context of the callee contract.

After the backing up, Pluto generates the ICPC for call inputs. Formula 8 describes the logic when deduce the input variable's ICPC.

$$\frac{\{p\}\ p := load(a)\ \{\hat{p}\}, \{C_a\}\ fp = ap\ \{C_i\}}{\{p, C_a\} \mapsto \{\hat{p}, C_i\}} \tag{8}$$

The symbol *p* and $\hat{p}$ in the formula mean the parameters of the called functions. We use *ap* and *fp* to represent 'actual parameters' and 'formal parameters' respectively. (C_a) and (C_i) represent the set of constraints before and after processing the call parameters. The logic presented by the formula is that when a function call: Pluto first finds out the variable refer to the formal parameter by checking the destination of the memory loading operation. The loading operation is represented as $load(a)$ in the formula where a denotes the memory address of the actual parameters. Then, Pluto simply add a constraint: *fp*==*ap* to the constraints set (C_a). This process can be described as a mapping from p with constraints C_a to $\hat{p}$ with constraints C_i. For the motivating example, Pluto first finds that the variable 'target' is the formal parameter of the call operation. Then, it loads the value of that variable from the memory with the opcode MLOAD.

$$\{C_i\}\ C_i.append(C_f)\ \{ICPC_{in}\} \tag{9}$$

During the execution of the call, the limits to the variable *fp* can be added to the initial constraints. This is described as formula 9, where C_f denotes the constraints of *fp*. The integrated constraints of *fp* is defined as the ICPC for call inputs which is represented as the symbol $ICPC_{in}$. In the motivating example, the *ap* is called 'currentTarget', so Pluto will add a constraint as: 'target == currentTarget' in the ICPC.

ICPC deduction for Call Outputs. After getting the proper constraints for call inputs, Pluto executes the called function. For all reachable return states, Pluto stores the return value into the inter-contract memory. When the call finishes, Pluto deduce the ICPC for the call outputs. The return value of the call operation has three forms: a real number, a variable and an expression.

$$\frac{\{O\}\ t := type(O)\ \{O_v\}}{\{C_r\}\ res = O_v\ \{ICPC_{out}\}} \tag{10}$$

If the return value is of the first two forms, the deduction of the ICPC for the call outputs is shown in formula 10. O refers to the output of the call, and O_v represents the type of the outputs is a real number or a variable. C_r denotes the constraints of the return value. res denotes the variable used to receive the return value. The formula's logic defines the ICPC for call outputs as the C_r together with a simple constraint 'res = O_v'.

If the return value is of an expression form, we deduce the ICPC for outputs as formula 11. E represents the set of expressions. Function $extract$ is used to extract the constraints of variables in expression O. The extracted constraints are named as C_e.

$$\frac{\{O\}\ t := type(O)\ \{t \in E\}, \{O\}\ extract(O)\ \{C_e\}}{\{C_e\}\ merge(C_e)\ \{C_r\}, \{C_r\}\ res = O\ \{ICPC_{out}\}} \tag{11}$$

Function $merge$ is used to get a union set of all extracted variables' constraints. The ICPC of this situation is deduced by adding the constraint: 'res = O' to C_r. In the motivating example contract, here Pluto will add such constraint to the ICPC: 'threshold == base_threshold OR threshold == currentTarget*100'

After the call finishes, Pluto will restore the original states. This process is described as formula 12. Function res can restore the original stack and storage states.

$$\{\mu\}\ \mu := res(\mu_{sta}, \mu_{sto})\ \{\hat{\mu}\} \tag{12}$$

With the ICPC for call outputs, Pluto can explore the contract under the original context and check for path reachability properly.

4.3 Bug Validation

During the ICFG exploring, Pluto checks whether the vulnerabilities exist in the current path. First, Pluto uses the constraint solver to check if the current path is reachable. For each feasible path, Pluto uses predefined rules to identify if the path is vulnerable. For each vulnerability, Pluto generates a report with detailed information and brief revision advice.

According to statistics [15], integer over/underflows, timestamp dependency, and reentrancy account for 97.64% of CVEs assigned to Ethereum smart contracts. So we designed Pluto with the detectors for these three kinds of bugs. However, according to some previous work [7], [10], the vulnerability detection rules can be easily added for contract analyzers.

Integer Overflow and Underflow Integer overflow and underflow is a common type of vulnerability in Ethereum smart contracts. Pluto checks all the numeric opcodes and finds out whether the result is improper (For example, if the result of an 'ADD' opcode is less than each operand). Then Pluto uses the AST of the contract to address the line numbers of all the mathematical expressions. In this way, Pluto can filter out all the numeric opcodes related to numerical processes and eliminates plenty of false positives. However, if the source code is missing, Pluto cannot filter false positives for now. In the future, we will analyze the difference between 'ADD' and 'SUB' opcodes in arithmetic operations and those in address offset calculation and filter the results on bytecode level. The revision advice of this kind of vulnerability is to use SafeMath library [24]. The other work [6], [8], [25] tries to detect this vulnerability by observing the operands and result of numeric opcodes('ADD','MUL','SUB', etc.). However, these opcodes are not only used in numerical processes but also used in calculations related to machine state, such as the calculation of storage offsets. This leads to plenty of false positives. Pluto inherited the algorithms used in the prior work. In addition, Pluto checks whether the operands of such opcodes are consist of variables from the smart contract source code.

Timestamp dependency In Ethereum, miners can modify the timestamp within certain limits [26]. Due to this fact, it is dangerous if the contract uses the block timestamp as a part of the conditions for token transferring. Here the token means both ERC20 tokens and ETH. Pluto checks two invariants to detect this vulnerability: the first is whether the opcode 'TIMESTAMP' is used in a condition expression. The second one is whether there is a token transfer operation in the expression with such opcode. The revision advice for this kind of vulnerability is to avoid using a timestamp as a condition to make some critical operations.

Oyente [6] detects this bug with the same methods as Pluto. SODA [27] and DEFECTCHECKER [28] detect this kind of vulnerability with only the first invariant of Pluto.

Reentrancy Reentrancy vulnerability first appeared in 'The DAO' attack in 2016 [2]. Grossman, S. et al [29] define it as a non-effective callback free contract. This vulnerability can let the attackers call back into the contract. The transfer of digital assets often accompanies this process. The attackers can steal a huge amount of tokens from the vulnerable contract. Figure 6 shows an example of reentrancy bugs. The contract called by Example can call back to it again since the balance of the callee contract has not yet been set to zero. The method Pluto uses to detect this vulnerability is to trace the unlimited-gas call operation with a non-zero value. Besides, if an external parameter defines the call address, Pluto reports it as a reentrancy bug. The revision advice for this bug is to use the function 'transfer' or 'send' when transferring tokens. With the 'transfer' or 'send' function, the fallback function is limited to the '2300' unit of gas, which is insufficient to initiate a new call.

```
contract Example{
    mapping (address => uint) balances;
    function withdrawBalance (){
        msg.sender.call.value(
            balances[msg.sender])();
        balances[msg.sender] = 0;
    }
}
```

Fig. 6: A contract with a reentrancy bug. The attacker can call back to the Example contract continuously and steal the tokens.

SODA and DEFECTCHECKER detect this vulnerability by tracking the money transfer without gas limits. Securify [11] uses a more loose way to detect this bug, but such flaws may not be used by attackers under any condition and are not security vulnerabilities strictly.

Support other types of vulnerabilities in Pluto. Similar to the bug validation methods listed above, Pluto can collect semantic information and inter-contract constraints related to other types of vulnerabilities. With the help of this information, Pluto can trace whether other vulnerabilities cause a malicious operation under the inter-contract scenarios. For example, to detect the 'Unchecked External Calls' bug defined in [30], Pluto can check whether there is a call operation with a gas limit of 2300 in the blocks in the ICFG. If there are such blocks, Pluto checks if they may jump to a block that throws exceptions. As for the detection of inconsistent behaviors of cryptocurrency tokens defined in [31], based on Pluto, one can add a trace recorder to store the symbolic execution trace of the contract. Then, he can locate the core data structure between the caller and callee contracts according to the ICFG. Finally, with the collection of ICPC, one can fetch token behaviors and compare them with each other to find inconsistent bugs. We will expand the bug types that Pluto supports in the future.

4.4 Features of Pluto.

As a symbolic execution tool, Pluto may have some abstraction which may affect its features, such as soundness, precision, and limitations.

In terms of soundness, the loop structure may lead to a very deep path and cause an explosion of the time taken by the constraint solver. Pluto approximates this by setting a depth to limit the biggest number of the length of a path. If the path is deeper than the setting, Pluto will stop explore it. This approximation is an under-approximation, but this is necessary to avoid symbolic execution from being stuck. Oyente and Mythril use this setting as well. In terms of precision, without the accurate information of the current blockchain, Pluto may not fetch the correct state of the block when dealing with related opcodes such as 'BLOCKHASH', 'NUMBER' etc. This will make Pluto not precise when dealing with contracts using such opcodes. However, this may not affect the execution process because all the results

of these opcodes will be represented as a symbol with no limits and stored in the stack. In terms of limitations, the time limit of Z3 may be the main limitation of Pluto. Since some complex constraints may cost lots of time to be solved, some paths may not be considered available. We will try to optimize all these features for Pluto in the future.

5 Evaluation

We evaluate Pluto to see whether it can detect vulnerabilities hidden in inter-contract scenarios efficiently and effectively. During the evaluation, we try to address the following research questions:

- **RQ1:** How does Pluto perform on detecting inter-contract vulnerabilities compared to state-of-art tools?
- **RQ2:** Does Pluto's strategy has any side effects on intra-contract vulnerabilities detection?
- **RQ3:** Can Pluto find vulnerabilities in real-world contracts in a relatively short time?

5.1 Dataset and Environment Setup

We used three datasets for the evaluation. The first dataset[3] contains 150 contracts with 150 inter-contract vulnerabilities. This dataset is constructed by summarizing the contract call scenarios from real contracts and adding common vulnerability patterns under those scenarios. We manually injected 50 reentrancy vulnerabilities, 50 timestamp dependency vulnerabilities, and 50 integer overflow and underflow vulnerabilities. This dataset is used to answer the first research question. The code other than the vulnerability patterns we inserted are classic contract code patterns, and these simple patterns do not contain vulnerabilities. To assess whether Pluto's strategy for inter-contract vulnerability detection has any side effects on intra-contract vulnerabilities, we use the contracts from SolidiFI dataset [18]. There are 898 manually injected intra-contract vulnerabilities in total in this dataset. To be specific, there are 121 reentrancy vulnerabilities, 176 timestamp dependency vulnerabilities, and 601 integer overflow and underflow vulnerabilities. We know in priori that all the labeled bugs in this dataset are intra-contract. However, this dataset may contain vulnerabilities except for the inserted ones. This indicates that the vulnerabilities reported by the evaluating tools in other locations are not always false positives. The way we calculated the false positives on this dataset is by manually checking all the reported vulnerabilities. The third dataset we select for evaluation is taken from the wild contract dataset provided by SmartBugs [19]. The dataset contains 8,173 solidity files with 39,443 real-world smart contracts. This dataset is taken from Etherscan [32], a leading block explorer and analytic platform for the Ethereum blockchain.

We compared Pluto with several state-of-art tools: Oyente, Mythril, ILF, Securify and Clairvoyance. The contract compiler we used is solc 0.4.26 [33]. We ran our experiment on a 64-bit machine with 16 cores(Intel(R) Xeon(R) Gold 5217). The operating system of the machine is Ubuntu 20.04.1 LTS and the main memory is 256 GB. We use solc 0.4.26 because most of the contracts in the SmartBugs dataset are developed several years ago. A high version compiler may have errors when compile them. However, Pluto can also work with more modern versions of the compiler.

In order to prevent the tools from getting stuck when detecting specific contracts, we follow the same parameter setting as the primary work. Specifically, the timeout for Z3 in Pluto and Oyente is set to 1 second, and the depth limit of both tools is set to 100. The depth limits the search depth during the symbolic execution. The timeout of Mythril is set to 300 seconds. The maximum transaction amount is set to 1000 for ILF. For Securify, we set the timeout as 300 seconds as well.

5.2 Pluto on Inter-Contract Vulnerabilities

To evaluate the performance of Pluto on detecting inter-contract vulnerabilities, we tested all the tools on 150 contracts with 150 inter-contract vulnerabilities. The results are shown in Table 2.

TABLE 2: Inter-contract vulnerabilities reported by Pluto and other five tools on 150 contracts. 'RE' represents reentrancy, while 'IO' and 'TD' mean integer overflow/underflow and timestamp dependency respectively. 'CV' refers to the tool Clairvoyance.

	Pluto	Oyente	Mythril	Securify	ILF	CV
RE	40	5	5	0	0	23
IO	40	5	5	N/A	N/A	N/A
TD	40	5	5	N/A	0	N/A

In consequence, Pluto can detect 80% of the injected inter-contract vulnerabilities. The vulnerabilities that Pluto failed to detect is due to the unreachable address of the callee contract. For example, the contract shown as the Fig 7 calls a function named 'ethBalance' in another interface 'F2mInterface'. The instance of that interface is named as 'f2mContract' in the code. However, as line 5 shows, the address of 'f2mContract' is given by the parameter of the function 'joinNetwork'. Pluto has no idea where the code

```
1  contract Bank{
2    function joinNetwork(address[6] _contract){
3      ...
4      // bugs here
5      _totalSupply += amount;
6      _balances[account] += amount;
7      ...
8    // f2mContract's address is passed by users
9      f2mContract = F2mInterface(_contract[0]);
10     f2m= 0xdfadfds....;
11     ...
12   }
13   function getDivBalance(address _sender)
14     public view returns(uint256){
15   // Pluto cannot know the address of callee
16     uint256 _amount =
           f2mContract.ethBalance(_sender);
17     if (_amount < 3000){
18       // some bugs here;
19     }
20     return _amount;
21   }
22 }
```

Fig. 7: A contract with a contract call. The called contract's address cannot be fetched. Pluto don't support the inter-contract bug detection on this contract.

3. The dataset with 150 contracts is available at: https://github.com/PlutoAnalyzer/pluto/tree/main/ManualSet.

can be found exactly. Other tools cannot handle this situation either. A possible solution may be that leveraging the history transactions. We will discuss this in more detail in Section 6.

Clairvoyance supports the detection of inter-contract reentrancy bugs as well. However, it may miss some real bugs due to the misuse of its path protective techniques(PPTs). For example, some contracts have a reentrancy bug after an irrelevant object's access control statement(PPT1 in Clairvoyance). Clairvoyance will consider these contracts as safe ones by misidentifying the protection techniques. Though all other tools do not support the detection of such vulnerabilities, they can also occasionally detect some of the vulnerable contracts with the wrong result of the call operation.

```
1  contract Test1{
2    uint public goal = 5000;
3    function getGoal() public returns(uint){
4      return goal;
5    }
6  }
7  contract Test2{
8    function test_2(Test1 t1, uint b)
9      public returns(uint){
10     uint goal_ = t1.getGoal();
11     // Oyente gets a very large goal_
12     if(3000 < goal_){
13       b += goal_;
14     }
15     return b;
16   }
17 }
```

Fig. 8: A contract with integer overflow vulnerability hidden in inter-contract scenarios. Oyente can report this vulnerability occasionally.

Fig 8 shows an example of contracts with inter-contract related vulnerability that can be found by Oyente. There is an integer overflow vulnerability at line 13 in the contract. Although Oyente cannot get the value of goal_ correctly, which is 5,000 in this case, it set goal_ as a very large value which is the same as the signature of function 'getGoal()' which equals to 'b97a7d24'. This value is 3111746852 in decimal format. The large value, however, is larger than 3,000 which gives Oyente a chance to enter the branch at line 13. As a result, Oyente can report this vulnerability though it is related to inter-contract scenarios.

Answer to RQ1: Pluto is effective on detecting inter-contract vulnerabilities. Compared with other tools, Pluto can detect the inter-contract vulnerabilities more accurately with no false positives and false negatives.

5.3 Side Effects on Intra-Contract Bugs

In order to find out whether there is a side effect of Pluto on detecting intra-contract bugs, we disabled the inter-contract capabilities of Pluto by deleting the related code snippets in Pluto and only reserving the symbolic execution code. Table 3 shows the result of Pluto and other tools for detecting intra-contract vulnerabilities. As the result shows, Pluto reported 681 valid vulnerabilities on 100 contracts, which is the best of all the tools. Pluto can report the most true-positive reentrancy vulnerabilities and integer overflow and underflow vulnerabilities of all the tools. On the detection of reentrancy bugs, Pluto can improve the highest recall by 90.08%(119/(2+119) - 10/(111+10)). As for timestamp dependency vulnerabilities, Pluto can report more real vulnerabilities than Oyente and ILF. However, Pluto reported 31 fewer real vulnerabilities than Mythril. The main reason for the false negatives is the setting of the Z3 timeout and system timeout of Pluto. If we give more time to Z3 and Pluto to analyze the contract, more vulnerabilities can be detected.

TABLE 3: Vulnerabilities reported by Pluto and other five tools on 100 contracts with 898 intra-contract vulnerabilities.'TP' refers to 'true positives', while 'FP' and 'FN' represent 'false positives' and 'false negatives' respectively. 'N/A' indicates that the tool does not support the detection of such type of vulnerability.

Tools	Reentrancy			Integer Overflow and Underflow			Timestamp Dependency		
	TP	FP	FN	TP	FP	FN	TP	FP	FN
Pluto	119	0	2	464	14	137	109	0	67
Oyente	110	0	11	172	487	429	42	1	134
Mythril	77	61	44	203	115	398	140	124	36
Securify	116	9	5	N/A	N/A	N/A	N/A	N/A	N/A
ILF	10	0	111	N/A	N/A	N/A	68	0	108
CV	97	75	24	N/A	N/A	N/A	N/A	N/A	N/A

In terms of false positives, Pluto reports no false alarms on reentrancy vulnerabilities and timestamp dependency vulnerabilities. Clairvoyance reports 75 false positives on reentrancy bugs. These false alarms mainly root in the incompleteness of PPTs in Clairvoyance. For example, when encountered with some complicated path conditions in access checking, Clairvoyance's lightweight symbolic analysis always fails to give an accurate result. For Mythril and Oyente, the false alarms are always due to the inaccurate definition of the vulnerabilities. For example, the function 'send' will not lead to a reentrancy bug. However, Mythril may consider the contract with this function as a vulnerable one. As for the integer overflow and underflow vulnerabilities, Pluto reports the fewest false positives of the tools. In the detection of this bug, Pluto can improve the highest precision by 70.97% (This can be calculated by (464/(14+464) - 172/(487+172))). All the false positives reported in this dataset are due to the lack of judgment of whether the bug is in a 'safe environment'.

For example, the contract snippets in Fig 9 are taken from one of the contracts in our dataset. Pluto considered this function contains an integer overflow vulnerability. This

```
1  function calcDynamicCommissionBegin(
2      uint256 index,
3      uint256 length
4      ) external onlyOwner {
5      // Pluto reports an overflow below
6      for(uint i=index;i<(index+length);++i) {
7        User storage user =
             userMapping[addressMapping[i]];
8        user.calcDynamicCommissionAmount = 0;
9      }
10 }
```

Fig. 9: An example of false positive reported by Pluto.

function is used to initialize the commission amount for

each user of this DApp. With no constraints of variable 'index' and 'length', the add operation at line 6 may overflow. The conditions to trigger this overflow are that the input value of 'index' and 'length' need to be very large. The function 'calcDynamicCommissionBegin', however, is limited by a modifier 'onlyOwner'. This modifier requires the caller of this function to be the owner of the contract, that is, a fixed address that deployed the contract on Ethereum. This address will not attack a contract deployed by itself, so this function is considered as executing in a "safe environment". However, it is tough to judge whether such a "safe environment" exists. This needs some semantic analysis which the current execution model of Pluto does not support. We will try to eliminate this kind of false positives in the future.

Answer to RQ2: Pluto is also effective on detecting intra-contract vulnerabilities. Compared with other tools, Pluto can improve the precision by up to 70.97% and improve the recall by up to 90.08% .

5.4 Pluto on Real-World Smart Contracts

We then evaluated Pluto and the other four tools on 8,173 real-world solidity files with 39,443 smart contracts. The results are shown in Table 4 and Table 5. For each tool, we calculated the total number of vulnerabilities, the inter-contract vulnerabilities, and the vulnerabilities that can only be found by this particular tool. All the vulnerabilities we listed in the table 4 are true vulnerabilities without false positives. The result shows that Pluto can find the most vulnerabilities in real-world contracts. Moreover, Pluto can detect 36 inter-contract vulnerabilities. In contrast, Oyente and Mythril can find 12 and 16 inter-contract vulnerabilities, respectively. We have explained the reason why they can find such vulnerabilities occasionally in the section 4.2. After manual comparison, we find that Pluto can find 47 vulnerabilities that other tools cannot find. 38 of them are of integer over/underflow type, and the other 9 are of timestamp dependency type.

The results in Table 5 show the false positives of each tool on this dataset. We have explained the reason why Pluto has false alarms in Section 5.3. According to the results, Pluto can report the least false positives on detecting reentrancy and integer overflow compared with the static and analysis tools. ILF can avoid false positives of other tools because the fuzzing technique can actually execute the contracts. As for the timestamp dependency bug, Pluto reports more false alarms than Oyente. The reason is that Pluto can support external calls. Meanwhile, Pluto imports more false positives from the callee contract side.

We now give two cases to describe the inter-contract vulnerabilities found by Pluto. The first case is taken from the contract deployed at address: '0x08283bd008112266568Bceffe13BB6c059Ae7A8A'. The contract is reported with a timestamp dependency vulnerability by Pluto. As Fig 10 shows, the function 'distribute' allows each pusher to buy the tokens released by H4Dcontract. However, to keep the pusher from getting too greedy, the function checks whether the current pusher buys the tokens too frequently. As line 4 to line 7 shown in Fig 10, after each token purchase, the pusher can only buy again after 100 other pushers have bought it and 1 hour has passed.

```
1  function distribute(uint256 _percent)
2     public isHuman()
3  {  ...
4     if ( pushers_[_pusher].tracker
5         <= pusherTracker_.sub(100) &&
6     // pusher is greedy: wait your turn
7     pushers_[_pusher].time.add(1 hours) < now )
          {
8     // pusher is greedy: not even been 1 hour
9       ...
10      // the token is sold to the pusher
11      H4Dcontract_.sell(
12        H4Dcontract_.balanceOf(address(this)));
13    }
14    ...
15 }
```

Fig. 10: A real-world contract with timestamp dependency vulnerability in inter-contract scenarios. Pluto can detect this vulnerability while other tools failed to.

However, miners can roughly modify the timestamp of the block by 900 seconds [34], while other miners will still accept the block. Due to this fact, malicious miners can always buy tokens in advance. This will significantly affect the token's value, and this greed will also cause losses to the token issuer. Pluto first fetched the code of 'H4Dcontract' from Ethereum and found that the selling process can be significantly affected by the timestamp of the current block. While other tools cannot properly call the 'sell' function in contract 'H4Dcontract' and failed to report this vulnerability.

The second case is taken from the contract deployed at address: '0x8888882056160e5ff4a0f26607d4a05bc506ca8c'. This contract contains an integer overflow vulnerability in inter-contract scenarios. Fig 11 shows the related code snippet. 'Lottery' is a game contract. Players can purchase lottery tickets to win the prize.

Function 'buyFor' shown in Fig 11 is used to sell the tickets to players. Players should spend at least the price of one ticket. The current price of a ticket is achieved by calling the function 'getTPrice' from the library 'Helper'. The function calculates the current ticket price. As line 23 shows, the base ticket price is represented as 'SLP'. The current price is the sum of the base price and the incremental price decided by the number of the sold tickets.

The overflow occurs at line 11 in the contract. The value of 'curRSalt' can be achieved by tracing the transaction history. A malicious player can buy a ticket with a constructed '_sSalt' to set the value of 'curRSalt' to zero. The variable 'curRSalt', together with the block number, is used to generate a seed that determines the lottery winner. If the value of 'curRSalt' is set to 0, the current block number will become the only basis for selecting the winner. However, it is unsafe to use block numbers to calculate the time [26]. Malicious miners can get bonuses illegally with this vulnerability.

Another finding from the Table 4 is that the percentage of the inter-contract vulnerabilities is relatively low (with 7.98% of the total found bugs are inter-contract ones.) The reason is that most contracts in SmartBugs dataset are tokens that can rarely make an external call. Table 6 shows the number of the contracts that call each other in SmartBugs and 100 random contracts deployed on Ethereum,

TABLE 4: The results of vulnerabilities found by Pluto and other five tools on 39,443 contracts. 'Total' refers to the total number of the vulnerabilities while 'IC' means the number of inter-contract vulnerabilities. 'Only' represents the vulnerabilities that can only be found by current tool.

Type	Pluto			Oyente			Mythril			Securify			ILF			CV		
	Total	IC	Only	Total	IC	Only	Total	IC	Only	Total	IC	Only	Total	IC	Only	Total	IC	Only
RE	8	0	0	8	0	0	7	0	0	2	0	0	3	0	0	6	0	0
IO	202	9	38	160	2	0	36	1	0	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
TD	241	27	9	118	10	0	195	15	0	N/A	N/A	N/A	97	0	0	N/A	N/A	N/A

TABLE 5: The results of false positives found by Pluto and other five tools on 39,443 contracts.

	Pluto	Oyente	Mythril	Securify	ILF	CV
RE	99	111	28145	284	28	124
IO	4593	6617	9887	N/A	N/A	N/A
TD	828	509	7401	N/A	250	N/A

```
contract Lottery{...
function buyFor(string _sSalt,address _sender)
  public payable buyable(){
  uint256 _salt = Helper.stringToUint(_sSalt);
  uint256 _ethAmount = msg.value;
  uint256 _ticketSum = curRticketSum;
  // need to get the correct ticket price here
  require(_ethAmount >= Helper.getTPrice(
    _ticketSum),"not enough to buy 1 ticket");
  // overflow occurs here
  curRSalt = curRSalt + _salt;
...}
}
library Helper{
  function getTPrice(uint256 _ticketSum)
    public pure returns(uint256){
      // ZOOM is 1000, PDIVIDER is 3450000
      uint256 base = (_ticketSum +
          1).mul(ZOOM) / PDIVIDER;
      uint256 expo = base;
      expo = expo.mul(expo).mul(expo); // ^3
      expo = expo.mul(expo); // ^6
      // SLP is 0.002 ether, PN is 777
      uint256 tPrice = SLP + expo / PN;
      return tPrice;
    }
}
```

Fig. 11: A real-world contract with integer overflow vulnerability under inter-contract scenarios. Pluto can detect this vulnerability while other tools failed to.

which recent transactions are based on. The result shows that external calls are relatively rare in SmartBugs with a percentage of 1.17%. While, in the recent contracts, the percentage has been increased to 86%. This indicates that a tool aims at detecting inter-contract bugs may be more and more important. In addition, as we illustrated in Section 5.2, Pluto does not support external calls without a fixed callee address. Though we have no ground truth about the false negatives of Pluto on SmartBugs dataset, according to the results in Table 6, situations which Pluto cannot resolve are really rare. Only 0.16% of the SmartBugs contracts and 1% of the recent contracts have such situations.

In order to evaluate the efficiency of Pluto in finding bugs, we calculated the time overhead of all the tools on this dataset. The results are shown in Table 7. All the numbers are calculated by the command-line tool 'time' [35] in Linux system. All these numbers are in seconds. On average, Pluto costs 92.69% less time than Mythril. Specifically, we recorded Pluto and Mythril's time cost for each contract, and at least Pluto can cost 46.79% less time than Mythril (A contract costs Pluto 5.6 seconds and Mythril for 10.4 seconds). At the highest, Pluto can cost 92.69% less time. (Another contract costs Pluto 2.7 seconds and Mythril for 36.5 seconds.) Mythril is slower than Pluto because it performs taint analysis and other operations in addition to symbolic execution. Securify is slower because the datalog analysis takes lots of time with complex rules.

TABLE 6: Contracts with external calls as well as the calls without a fixed address in SmartBugs dataset and 100 random contracts which recent transactions are based on.

	SmartBugs		Recent contracts	
	Number	Percentage	Number	Percentage
External Calls	96	1.17%	86	86.00%
External Calls without fixed address	13	0.16%	1	1.00%

TABLE 7: Time overhead of Pluto compared with state-of-arts. The numbers in the table are in seconds.

	Pluto	Oyente	Mythril	Securify	ILF	CV
Avg	16.9	15.8	231.8	40.4	31.9	8.1
Min	0.3	0.3	1.3	0.9	2.4	0.4
Max	192.1	221.1	5975.2	689.3	63.3	121.5

ILF is slower because it needs to learn from a model to generate new inputs for each iteration. Clairvoyance depends on light-weight data/control flow analysis, which results in faster scanning. However, as illustrated in table 3, Clairvoyance can report many false positives without a thorough analysis. As for Oyente, Pluto costs 1.1 more seconds on average for a contract. This is due to the construction of the ICFG and ICPC and is tolerable for users.

Answer to RQ3: Pluto can find 451 confirmed vulnerabilities in real-world contracts. Among them, 36 vulnerabilities are related to inter-contract scenarios. Pluto needs 16.9 seconds to analyze a contract on average which is competitive compared to other tools.

5.5 Pluto without Source-level Filtering

In Section 4.3, we have mentioned that Pluto will leverage the AST of smart contracts to filter the false positives caused by the arithmetic opcodes used to calculate offsets of stack address. In order to evaluate the performance of Pluto without AST, we conducted an experiment on both SolidiFI and SmartBugs dataset. The results are shown as Table 8.

The results show that without the filtering, Pluto has a high false-positive rate, just like Oyente and Mythril on overflow bug detection. This indicates that the filtering strategy is useful. However, if the source code is missing,

TABLE 8: False positives of integer overflow bugs reported by Pluto without AST and other symbolic execution tools on both SolidiFI and SmartBugs dataset.

Tools	Pluto without AST		Oyente		Mythril	
	FP	FP-rate	FP	FP-rate	FP	FP-rate
SolidiFI	487	51.21%	487	73.90%	115	36.16%
SmartBugs	6571	97.02%	6617	97.64%	9887	99.64%

Pluto cannot filter these false positives. In the future, we will analyze the difference of 'ADD' and 'SUB' opcodes in arithmetic operations and those in address offset calculation and try to filter the results on the bytecode level.

6 Discussion

In this section, we will discuss some limitations that can threaten the performance of Pluto.

Contract Call without a Fixed Contract Address. Generally, smart contract always defines the contract address it would like to call in the current contract. Before executing the call opcode, EVM will put the callee's address into the stack. During the symbolic execution stage, Pluto can get the address of the callee contract from the content in the stack when encountered with a call operation. Pluto will fetch the bytecode of the called contract with the help of the API provided by Ethereum. Based on the decompiler provided by Etherscan, Pluto can get the signature of the called function and commit the external call. We will also enhance Pluto with SigRec [36] in the furture to get the signature more accurately. However, some real-world smart contracts do not point out the address of the called contract just like the contract showing in Fig 7. In an actual transaction, the user knows the exact address to pass in as the parameter of the function. However, it cannot be fetched by symbolic execution tools automatically without the knowledge of the actual transactions' information.

Thus, a possible way to solve this problem is to track the historical transactions of this contract. The transactions information which contains the parameters' values may indicate the correct address of the callee contract. With an analysis of them, it may be possible to retrieve the called contract address based on the transaction records. Besides, the approached provided by the work [37] can lift the EVM bytecode to provide a high-level IR, and the IR can model the external call more precisely. With the help of its memory model, Pluto can handle external calls without a fixed address. We will implement this feature on Pluto in the future.

Comparison with the 'naive' way to support inter-contract scenarios. An obvious and naive way to support inter-contract vulnerabilities using a tool that only supports intra-contract analysis would be to resolve the callees directly. Thus, the comparison between Pluto an static and symbolic execution tools seems to be unfair. However, such so-called 'naive' ways do not exist for several reasons: (1) For the static analysis tools, it can be hard to get the proper function to call. Match the function with the same signature and analyze the callee contract's code can be very hard without rebuilding the models and the IR used by the current static tools. (2) For the symbolic execution tools, in order to call another contract, the tool needs to construct a correct model for call operations. This arises some questions like: How to find and pass the correct parameters? Where to put the results? How to connect the path constraints? All these questions are necessary to enable symbolic execution to support inter-contracts.

So, it can be said that for static analysis and symbolic execution tools, there are no such naive ways to correctly model the external calls. While Pluto leverages the construction of ICFG and the deduction of ICPC to model the external call process and makes a thorough analysis on this procedural. With the help of these techniques, Pluto can find more bugs with less false positives

Path Explosions. The first threat comes from the performance of the symbolic execution engine. Symbolic execution tools can suffer from path explosion problems when dealing with complex programs [38], [39]. Although smart contracts are simpler than traditional programs, the path explosion also arises when Pluto deals with contracts that have complex loops and large path depths. This will reduce Pluto's coverage of the contract and lead to false negatives. A good solution for this threat is to use a heuristics algorithm in the path finding process [40]. Besides, some researchers try to conduct symbolic execution in parallel to reduce the phenomenon [41], [42]. For example, GasChecker proposes parallelizing symbolic execution by tailoring it to the MapReduce programming model. We will try to leverage these techniques on Pluto later.

SMT Solver Configurations. Pluto leverages Z3 to solve the path constraints collected during the symbolic execution of contracts. However, we found that the time limit of Z3 may significantly influence the soundness of Pluto. If we give Z3 more time to seek for a solution of each constraint, Pluto can enter deeper path and may find more vulnerabilities there. However, if we don't extend the total execution time of Pluto, the time spent here can prevent the execution of subsequent code.

For example, we evaluated Pluto on a contract in SolidiFI dataset with a global timeout of 10 seconds. When we set the timeout of Z3 to 10 milliseconds, Pluto can report only one timestamp dependency bug. While when we set the Z3's timeout to 100 milliseconds, Pluto can report 8 timestamp dependency bugs. A good way to eliminate this problem is to evaluate the optimal parameter settings through extensive experimentation [43].

7 Related Work

With increasing attacks on Ethereum smart contracts [44], [45], [46], [47], many researchers have developed contract vulnerability detection tools. We will introduce them according to the techniques they use.

Static analysis tools can detect smart contract vulnerabilities without executing the contract. Most of them analyze contracts on an intermediate representation, like Securify [11] and MadMax [12]. Other tools, such as SmartCheck [21], translates contract source code into an XML-based intermediate representation and checks it against XPath patterns. A recent work named Clairvoyance [17] defines five path protection techniques to detect reentrancy vulnerabilities more accurately.

The symbolic execution technique is also widely used to detect vulnerabilities in smart contracts. Oyente [6] is considered the first tool using symbolic execution to find bugs in smart contracts. Based on Oyente, Maian [7] defines new rules to detect greedy, prodigal, and suicidal contracts. Osiris [25] is another tool based on Oyente aimed at finding

integer-related bugs. In the industry, Mythril [16] is a popular tool that combines symbolic execution and taint analysis.

Apart from the above techniques, fuzzing is also a useful way to detect vulnerabilities in programs [48], [49], [50], [51]. Many researchers apply fuzzing to smart contracts. ContractFuzzer [8] and sFuzz [9] imitate the actual execution environment of smart contracts and use different inputs to check whether a vulnerability can be triggered. Unlike the above work, ILF [10] uses imitation learning to generate high-quality seeds to cover some deep paths.

The main difference between these tools and Pluto is that most do not support vulnerability detection in the inter-contract scenario. The static analysis tools construct their analysis only based on a single contract. Although Clairvoyance supports the detection under cross-contract calls, it can report many false positives and negatives due to its incomplete definition of reentrancy bug and PPTs. As for the symbolic execution tools always fail to get the right path reachability information under smart contract calls. The fuzzing tools such as ContractFuzzer and sFuzz cannot enter into some branches in inter-contract scenarios. Though ILF's strategy can improve the seed quality in single contract analysis, it will fail under inter-contract calls.

8 Conclusion

In this paper, we propose Pluto for inter-contract vulnerability detection. Pluto uses an ICFG to gain semantic information from inter-contract scenarios. Besides, Pluto leverages a series of customized Hoare logic to deduct ICPC to check the execution path reachability correctly. We evaluate Pluto and five state-of-art tools on 150 labeled contracts and 39,443 real-word contracts. The result shows that Pluto is more effective than other tools in the detection of inter-contract bugs. Besides, Pluto finds 451 confirmed vulnerabilities in real-world contracts, 36 of which are hidden in inter-contract scenarios. Two of the bugs are assigned with CVE identifiers by US National Vulnerability Database. As for the detection of intra-contract bugs, Pluto also performs better by decreasing both false-positive and false-negative rates of other tools. In terms of time consumption, Pluto takes an average of 16.9 seconds to analyze a contract. We will expand Pluto with more vulnerability types in the future.

References

[1] SECBIT, "A disastrous vulnerability found in smart contracts of beautychain (bec)," https://medium.com/secbit-media/a-disastrous-vulnerability-found-in-smart-\contracts-of-beautychain-bec-dbf24ddbc30e, 2018, accessed September 6, 2020.

[2] S. Falkon, "The story of the dao — its history and consequences," https://medium.com/swlh/the-story-of-the-dao-its-history-and-\consequences-71e6a8a551ee, 2017, accessed September 6, 2020.

[3] S. Lee, "Blockchain smart contracts: More trouble than they are worth?" https://www.forbes.com/sites/shermanlee/2018/07/10/\blockchain-smart-contracts-more-trouble-than-they-are-worth/493735e623a6, 2018, accessed September 6, 2020.

[4] S. CHANG, "Ethereum smart contracts vulnerable to hacks: $4 million in ether at risk," https://www.investopedia.com/news/ethereum-smart-contracts-vulnerable-hacks\-4-million-ether-risk/, 2019, accessed September 6, 2020.

[5] E. Org, "Ethereum," https://ethereum.org/en/, 2020, accessed September 6, 2020.

[6] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[7] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018.

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

[9] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. Minh, "sfuzz: An efficient adaptive fuzzer for solidity smart contracts," *ArXiv*, vol. abs/2004.08563, 2020.

[10] J. He, M. Balunovic, N. Ambroladze, P. Tsankov, and M. T. Vechev, "Learning to fuzz from symbolic execution with application to smart contracts," *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.

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

[12] 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, pp. 1 – 27, 2018.

[13] T. Chen, Y. Zhu, Z. Li, J. Chen, X. Li, X. Luo, X. Lin, and X. Zhange, "Understanding ethereum via graph analysis," in *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, 2018, pp. 1484–1492.

[14] Z3Prover, "Z3," https://github.com/Z3Prover/z3, 2020, accessed September 22, 2020.

[15] S. So, M. Lee, J. Park, H. Lee, and H. Oh, "Verismart: A highly precise safety verifier for ethereum smart contracts," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1678–1694.

[16] ConsenSys, "Mythril," https://github.com/ConsenSys/mythril, 2018, accessed September 15, 2020.

[17] J. Ye, M. Ma, Y. Lin, Y. Sui, and Y. Xue, "Clairvoyance: cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, 2020, pp. 274–275.

[18] A. Ghaleb and K. Pattabiraman, "How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection," *arXiv preprint arXiv:2005.11613*, 2020.

[19] SmartBugs, "Smartbugs wild dataset," https://github.com/smartbugs/smartbugs-wild, 2020, accessed October 5, 2020.

[20] Ethereum, "Ethereum: A secure decentralised generalised transaction ledger," https://ethereum.github.io/yellowpaper/paper.pdf, 2020, accessed September 21, 2020.

[21] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2018, pp. 9–16.

[22] T. Chen, Z. Li, Y. Zhang, X. Luo, T. Wang, T. Hu, X. Xiao, D. Wang, J. Huang, and X. Zhang, "A large-scale empirical study on control flow identification of smart contracts," in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2019, pp. 1–11.

[23] Solidity, "Solidity compiler," https://solidity.readthedocs.io/en/v0.5.3/\installing-solidity.html, 2020, accessed September 28, 2020.

[24] jdourlens, "Using safe math library to prevent from overflows," https://ethereumdev.io/using-safe-math-library-to-prevent-from-\overflows/, 2020, accessed October 4, 2020.

[25] 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.

[26] SWC, "Block values as a proxy for time," https://swcregistry.io/docs/SWC-116, 2020, accessed November 10, 2020.

[27] 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 framework for smart contracts." in *NDSS*, 2020.

[28] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen, "Defectchecker: Automated smart contract defect detection by analyzing evm bytecode," *IEEE Transactions on Software Engineering*, pp. 1–1, 2021.

[29] S. Grossman, I. Abraham, G. Golan-Gueta, Y. Michalevsky, N. Rinetzky, S. 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, pp. 1 – 28, 2018.

[30] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen, "Defining smart contract defects on ethereum," *IEEE Transactions on Software Engineering*, pp. 1–1, 2020.

[31] T. Chen, Y. Zhang, Z. Li, X. Luo, T. Wang, R. Cao, X. Xiao, and X. Zhang, "Tokenscope: Automatically detecting inconsistent behaviors of cryptocurrency tokens in ethereum," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1503–1520. [Online]. Available: https://doi.org/10.1145/3319535.3345664

[32] Etherscan, "A block explorer for ethereum," https://etherscan.io/, 2020, accessed October 27, 2020.

[33] Ethereum, "Solidity compiler 0.4.26," https://github.com/ethereum/solidity/releases/tag/v0.4.26, 2020, accessed October 5, 2020.

[34] Eth, "Welcome to the ethereum wiki!" https://eth.wiki, 2020, accessed November 3, 2020.

[35] "Linux time command," https://linuxize.com/post/linux-time-command/, 2019, accessed at April 5th, 2021.

[36] T. Chen, Z. Li, X. Luo, X. Wang, T. Wang, Z. He, K. Fang, Y. Zhang, H. Zhu, H. Li, Y. Cheng, and X.-s. Zhang, "Sigrec: Automatic recovery of function signatures in smart contracts," *IEEE Transactions on Software Engineering*, pp. 1–1, 2021.

[37] S. Lagouvardos, N. Grech, I. Tsatiris, and Y. Smaragdakis, "Precise static modeling of ethereum "memory"," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–26, 2020.

[38] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[39] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution." in *NDSS*, vol. 16, no. 2016, 2016, pp. 1–16.

[40] K.-K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks, "Directed symbolic execution," in *International Static Analysis Symposium*. Springer, 2011, pp. 95–111.

[41] M. Staats and C. Păsăreanu, "Parallel symbolic execution for structural test generation," in *Proceedings of the 19th international symposium on Software testing and analysis*, 2010, pp. 183–194.

[42] T. Chen, Y. Feng, Z. Li, H. Zhou, X. Luo, X. Li, X. Xiao, J. Chen, and X. Zhang, "Gaschecker: Scalable analysis for discovering gas-inefficient smart contracts," *IEEE Transactions on Emerging Topics in Computing*, pp. 1–1, 2020.

[43] M. Ren, Z. Yin, F. Ma, Z. Xu, Y. Jiang, C. Sun, H. Li, and Y. Cai, "Empirical evaluation of smart contract testing: What is the best choice?" in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 566–579. [Online]. Available: https://doi.org/10.1145/3460319.3464837

[44] 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.

[45] T. Chen, X. Li, Y. Wang, J. Chen, Z. Li, X. Luo, M. H. Au, and X. Zhang, "An adaptive gas cost mechanism for ethereum to defend against under-priced dos attacks," in *International Conference on Information Security Practice and Experience*. Springer, 2017, pp. 3–24.

[46] S. Sayeed, H. Marco-Gisbert, and T. Caira, "Smart contract: Attacks and protections," *IEEE Access*, vol. 8, pp. 24 416–24 427, 2020.

[47] H. Chen, M. Pendleton, L. Njilla, and S. Xu, "A survey on ethereum systems security: Vulnerabilities, attacks, and defenses," *ACM Computing Surveys (CSUR)*, vol. 53, no. 3, pp. 1–43, 2020.

[48] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Transactions on Software Engineering*, vol. 45, no. 5, pp. 489–506, 2017.

[49] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2329–2344.

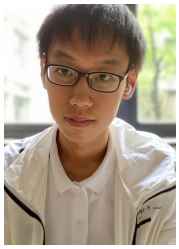
[50] C. Lemieux and K. Sen, "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 475–485.

[51] Y. Chen, Y. Jiang, F. Ma, J. Liang, M. Wang, C. Zhou, X. Jiao, and Z. Su, "Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 1967–1983.

Fuchen Ma received the BS degree in software engineering from Beijing University of Posts and Telecommunications, Beijing, China, in 2019. He is currently working toward the Ph.D. degree in software engineering at Tsinghua University, Beijing, China. His research interests include fuzzing testing and security of blockchain systems.

Zhenyang Xu received the bachelor degree in electronic science and technology from Zhejiang University, Hangzhou, China, in 2020. He is pursing a Ph.D. degree in computer science at University of Waterloo, Waterloo, Canada. His research interests include fuzzing testing and programming language.

Meng Ren received the BS degree in school of software engineering, SUN YAT-SEN University, Guangzhou, China, in 2019. She is currently working toward the Master degree in with school of software, Tsinghua University, Beijing, China. Her research interests are fuzzing technology and the security of blockchain systems.



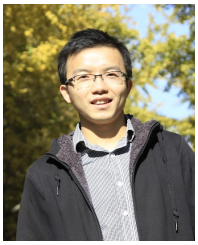
Zijing Yin received his BS degree in information security from China University of Mining and Technology, Xuzhou, China, in 2020. He is currently working toward a Master's degree in software engineering at Tsinghua University, Beijing, China. His research interests include Web security and the security of blockchain systems.

Yuanliang Chen received the BS degree in Nanjing University in 2017. In 2020, he received his Master degree in Tsinghua University, Beijing, China. His research interests including fuzzing technology and the architecture of blockchain systems. He is now working on the security of blockchain protocols.

Lei Qiao received his Ph.D. degrees in computer science from USTC Hefei, in 2007. He is a professor in Beijing Institute of Control Engineering. His research interests include operating system design and formal verification.

Bin Gu is a professor at Beijing Institute of Control Engineering. He received an M.Sc. in computation mathematics from Harbin Institute of Technology in 1994, and a Ph.D. in Electronics and Information Technology from Northwestern Polytechnical University in 2020. Professor Bin Gu has been working on embedded control system and real-time embedded software, his current research interests focuses mainly on trustworthy embedded system and software, self-adaptation software and Intelligent technologies for software engineering.

Huizhong Li received his Master degree in Peking University. He has worked for Tencent and Webank, and is the head of R&D of blockchain underlying platform at Webank. He is currently working on the blockchain technology and privacy computing. He led his team to develop FISCO BCOS, one of the most popular consortium chain systems in China. His research interests include distributed network, consensus protocol, system architecture and security.



Yu Jiang received the BS degree in software engineering from Beijing University of Posts and Telecommunications in 2010, and the PhD degree in computer science from Tsinghua University in 2015. He worked as a Postdoc researcher in the department of computer science of University of Illinois at Urbana-Champaign, IL, USA, in 2016, and is now an assistant professor in Tsinghua University. His current research interests include domain specific modeling, formal computation model, formal verification and their applications in embedded systems.

Jiaguang Sun received the BS degree in automation science from Tsinghua University in 1970. He is currently a professor in Tsinghua University. He is dedicated in teaching and R&D activities in computer graphics, computer-aided design, formal verification of software, and system architecture. He is currently the director of the School of Information Science & Technology and the School of Software in Tsinghua University.