Detecting Nondeterministic Payment Bugs in Ethereum Smart Contracts

SHUAI WANG*, The Hong Kong University of Science and Technology, China
CHENGYU ZHANG, East China Normal University, China
ZHENDONG SU, ETH Zurich, Switzerland

The term "smart contracts" has become ubiquitous to describe an enormous number of programs uploaded to the popular Ethereum blockchain system. Despite rapid growth of the smart contract ecosystem, errors and exploitations have been constantly reported from online contract systems, which has put financial stability at risk with losses totaling millions of US dollars. Most existing research focuses on pinpointing specific types of vulnerabilities using *known patterns*. However, due to the lack of awareness of the inherent *nondeterminism* in the Ethereum blockchain system and how it affects the funds transfer of smart contracts, there can be unknown vulnerabilities that may be exploited by attackers to access numerous online smart contracts.

In this paper, we introduce a methodical approach to understanding the inherent nondeterminism in the Ethereum blockchain system and its (unwanted) influence on contract payments. We show that our new focus on nondeterminism-related smart contract payment bugs captures the *root causes* of many common vulnerabilities without relying on any known patterns and also encompasses recently disclosed issues that are not handled by existing research. To do so, we introduce techniques to systematically model components in the contract execution context and to expose various nondeterministic factors that are not yet fully understood. We further study how these nondeterministic factors impact contract funds transfer using information flow tracking. The technical challenge of detecting nondeterministic payments lies in discovering the contract global variables subtly affected by read-write hazards because of unpredictable transaction scheduling and external callee behavior. We show how to augment and instrument a contract program into a representation that simulates the execution of a large subset of the contract behavior. The instrumented code is then analyzed to flag nondeterministic global variables using off-the-shelf model checkers.

We implement the proposed techniques as a practical tool named NPChecker (Nondeterministic Payment Checker) and evaluate it on 30K online contracts (3,075 distinct) collected from the Ethereum mainnet. NPChecker has successfully detected nondeterministic payments in 1,111 online contracts with reasonable cost. Further investigation reports high precision of NPChecker (only four false positives in a manual study of 50 contracts). We also show that NPChecker unveils contracts vulnerable to recently-disclosed attack vectors. NPChecker can identify all six new vulnerabilities or variants of common smart contract vulnerabilities that are missed by existing research relying on a "contract vulnerability checklist."

CCS Concepts: • **Software and its engineering → General programming languages**.

Additional Key Words and Phrases: Program Analysis, Smart Contracts, Security, Blockchain

ACM Reference Format:
Shuai Wang, Chengyu Zhang, and Zhendong Su. 2019. Detecting Nondeterministic Payment Bugs in Ethereum Smart Contracts. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 189 (October 2019), 29 pages. https://doi.org/10.1145/3360615

*This work is done while Shuai Wang was working at ETH Zurich.

Authors' addresses: Shuai Wang, The Hong Kong University of Science and Technology, China, shuaiw@cse.ust.hk; Chengyu Zhang, East China Normal University, China, dale.chengyu.zhang@gmail.com; Zhendong Su, ETH Zurich, Switzerland, zhendong.su@inf.ethz.ch.

2475-1421/2019/10-ART189
https://doi.org/10.1145/3360615

1 INTRODUCTION

Blockchain is a computing platform where computers run in a decentralized fashion and are coordinated by consensus among the underlying miners. Ethereum is the world's second largest cryptocurrency (behind Bitcoin) by market cap, and allows users to write and upload smart contracts, which are general-purpose programs that execute on the blockchain. Thus far, a variety of smart contracts applications have been launched on Ethereum, including financial services, supply chain systems, and games. Users of Ethereum can execute smart contracts by sending money to them. As of March 2019, the total value of Ethereum currency has reached over 15 billion US dollars.[1]

Despite this prosperity, investment in Ethereum is still considered speculative and unregulated. The market fluctuates frequently, and the security of Ethereum smart contracts is also of general concern. Due to strategic motives of contract developers (i.e., "time-to-market" requirements) and blockchain system constraints, smart contracts are generally perceived as prone to exploitation and security breaches. Massive attacks have been launched to exploit online smart contracts, causing severe threats to financial stability. For instance, the DAO attack successfully stole over 3.6M Ether and caused a hard fork (a rule-violating change in the blockchain) in order to revert the attack [Siegel 2016]. Even worse, smart contracts are difficult to update after release, which means that contract developers are mostly unable to ensure that their contracts will be iteratively fixed for security flaws. Hence, a large number of both known and unknown defects could exist in smart contracts (as has been confirmed by recent research and industrial hackers [evm 2018b; Code4Block 2018; Luu et al. 2016; SECBIT 2018; Tsankov et al. 2018]). These defects present a serious challenge to both contemporary security and blockchain research.

Among current efforts in this demanding field, static analysis is used to flag several specific types of vulnerabilities, given the source code or binary code of contract programs. Meanwhile, abnormal behaviors are monitored during online execution by instrumenting the Ethereum Virtual Machine (EVM). Most of these existing approaches utilize appropriate *predefined vulnerability patterns* which can only be found by experts [ConsenSys 2018; Jiang et al. 2018; Kalra et al. 2018; Luu et al. 2016; Rodler et al. 2018; Tsankov et al. 2018]. Hence, our capability of finding defects and identifying contract vulnerabilities is limited by the currently known patterns.

A systematic methodology is yet to be found to understand the *root causes* of many well-known vulnerabilities. This work aims at analyzing *nondeterminisms* in the smart contract execution context (e.g., unpredictable transaction scheduling, as discussed in [Sergey and Hobor 2017]; see Sec. 2.3 for the clarification of "nondeterminisms" in our research context) that cause unpredictable payments and potential financial loss. Rather than searching for likely buggy payments with the help of *predefined patterns*, our approach is unique and performs systematic modeling to expose various nondeterministic factors in the contract execution context — such nondeterminism indicates the *root cause* of common vulnerabilities and helps detect presumably exploitable payment bugs *without querying any patterns*.

To this end, we formulate the contract in an intermediate language and use information flow analysis to identify contract payments influenced by nondeterministic factors existing in the contract execution context. We aim to flag the usage of certain nondeterministic system properties and external call returns in contracts. More importantly, we aim to comprehensively identify program global variables under read-write hazards (and hence becoming "nondeterministic") due to the unpredictable transaction scheduling and external callee behaviors. To this end, we apply systematic instrumentation and promote the contract language with various runtime components: the enhanced language is able to model a large subset of the contract behaviors, and enables us to identify program global variables that may be nondeterministically updated.

The proposed techniques are implemented as a practical tool named NPChecker that analyzes the EVM bytecode compiled from the contract source code. From the 30K smart contracts that we used for evaluation (including 3,075 unique contracts, as per their `sha256sum`), NPChecker has

[1]As of this writing, the price of each Ethereum token, called Ether, is worth 162.3 US dollars.

successfully found 1,111 buggy contracts. Comparing with existing research in this field, we show that NPChecker reveals payment bugs in a highly precise manner (only four false positives in our manual inspection of 50 contracts) and therefore making bug confirmation and rectification much easier. We also show that NPChecker unveils subtle issues in real-world contracts that are unknown to existing research. NPChecker can identify all six recently-disclosed vulnerabilities or variants of common vulnerabilities that are missed by existing research relying on predefined bug patterns. In summary, we make the following contributions:

- We introduce and advocate a new focus on the inherent nondeterminism of Ethereum — the *root cause* of various common smart contract vulnerabilities. We strive to detect nondeterminism that lead to unpredictable funds transfer which are presumably vulnerable (referred as *payment bugs* in the rest of the paper) without using *any predefined patterns*. Our research captures a large set of payment bugs beyond the well-known ones: some subtle vulnerabilities disclosed in recent studies are as well exposed.
- We use information flow tracking as a unified approach to study how nondeterministic contract variables can affect funds transfer. We formulate the contract program in a well-defined language as the basis of the information flow tracking. To expose nondeterministic global variables, we propose systematic instrumentations to take runtime components (e.g., miner) that can potentially affect the contract execution into account. The instrumented language practically models contract execution and is linked with model checkers (Smack [Rakamarić and Emmi 2014]) to pinpoint contract global variables suffering from read-write hazards.
- We implement the proposed technique into a practical tool named NPChecker to detect payment bugs in EVM bytecode. Out of the 30K online contracts (of which 3,075 were unique) collected from the Ethereum mainnet, NPChecker successfully flagged 1,111 buggy contracts. Further manual investigation shows that NPChecker is high accurate.

2 BACKGROUND

2.1 Ethereum Blockchain and Transaction Pending Pool

Ethereum is a decentralized peer-to-peer network which maintain and secure a shared ledger called blockchain [Wood 2014]. Typical actors in Ethereum include miners and Ethereum accounts. Miners are computers residing on nodes of this peer-to-peer network to construct and maintain the underlying blockchain. Ethereum accounts are the primary "users" of Ethereum for various (commercial) purposes. Typically, each account could be controlled by private keys (i.e., "externally owned account") or controlled by its own code (see Sec. 2.2). Each account, indexed by a unique address, can interact with other accounts on Ethereum by creating transactions. Each transaction can be an arbitrary message, payment, or contract creation command.

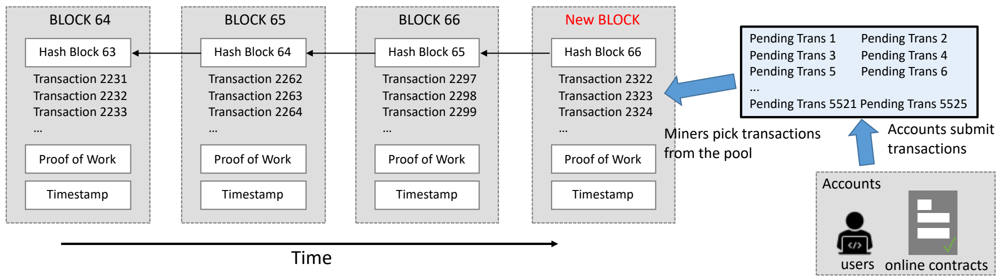

Fig. 1. The Ethereum blockchain and the transaction pending pool. Miners are free to include any transactions into the new block in any order, which entails a primary source of nondeterminism (see Sec. 2.3.2 for details).

As shown in Fig. 1, the ledger is organized as a hash-chain of blocks ordered by time, and each block contains a set of transactions. To create a new block, each miner selects certain transactions from a transaction pending pool and add them to a new block. Then, the miner starts to solve a cryptographic puzzle (i.e., "mining" [Nakamoto et al. 2008; Wood 2014]), and when a miner first solves the puzzle, it will broadcast its result across the entire network and the result is verified by other miners. Once consensus is reached, the winner miner will receive a reward and add its proposed new block to the blockchain.

Fig. 1 also depicts how a miner selects transactions from the pending pool. In general, when a transaction is created by an account, the sender can specify a limit to the gas they are willing to spend for the transaction (called the "gas price"). The transaction then waits in the transaction pending pool, until it gets picked and executed by an miner. Miners tend to pick valid transactions to fill a block in a way that maximizes profits. Therefore, transactions that are assigned higher gas prices are usually prioritized. After a miner successfully generates a new block, all the transactions in the block are appended to the end of the chain. Also, while the pending pool can contain plenty of transactions, the number of transactions that can be injected into one block is limited by the bounded maximum gas cost per block. Hence, a transaction may stay in the pool for a considerable amount of time before being executed.

2.2 Ethereum Smart Contracts

A smart contract is a program which defines and enforces agreements between Ethereum users. Smart contract programs (usually written in `Solidity`) are compiled into bytecode and deployed to the Ethereum blockchain. Fig. 2 shows how contracts (in terms of bytecode) fit into Ethereum. Ethereum virtual machine (EVM) is deployed to every node and executes contract bytecode. EVM bytecode has a very succinct representation that includes approximately 70 different opcodes for computations and communication with the underlying blockchain infrastructure. While the source code of smart contracts is usually not available, the EVM bytecode of every online contract is publicly accessible on the blockchain.

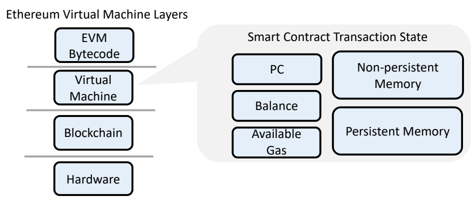

Fig. 2. Ethereum blockchain layers.

All contracts deployed on EVM can access its own account balance (i.e., the amount of Ether it has), its private storage, and its EVM bytecode. In general, there are two types of private storage a contract bytecode can access: the *non-persistent* storage that discards its content after one transaction is completed, and the *persistent* storage retaining a key-value store persisted across transactions. Every EVM instruction costs a certain amount of gas. In general, accessing non-persistent storage causes far less overhead (less than five gas) than accessing persistent storage (usually between 200 to 20,000 gas).

A Contract Example. Fig. 3 presents a sample contract `DigitalGoods`, which acts as an intermediary to purchase goods from `owner` for `user`, by taking certain amount of `fee` from the user's purchase. To deploy this contract, the contract owner schedules a contract creation transaction, including the contract bytecode. When the creation transaction is accepted, a unique address will be assigned for this contract, its persistent storage will be allocated (e.g., `fee`), and further initialized by calling the constructor function (`Intermediary()`). Note that for a contract transaction, the information of the sender (the amount of transferred Ether, sender's address, etc.) are all kept in variable `msg`. After acquiring the purchase fee which is publicly accessible (i.e., $10), users can submit the purchase transaction, by calling the `purchase` function, which issues a *payment* call (via `transfer`). Also, the `setFee` function can be used to adjust the fee, in case the transaction sender is the contract owner.

While the contract looks normal, the owner can indeed trap contract users, by forcing them to accept a higher fee. In fact, the malicious owner of this contract could actively monitor transactions in the pending pool and schedule a transaction towards the `setFee` function to increase `fee`. By raising the miner reward higher to the value of the user's transaction, the `setFee` transaction will be executed first (e.g., raise the purchase fee from $10 to $100), forcing the user to send less

```
contract Intermediary {
  uint256 public fee;
  address public seller;
  address public owner;

  function Intermediary() {
    owner = msg.sender;
// seller initialization is omitted
    fee = 10;
  }
```

A sample smart contract.

```
  function purchase() {
// msg.value is how much Ether was sent by user
// transfer pays (msg.value-fee) to the seller
    owner.transfer(msg.value - fee);
  }
  function setFee(uint256 _fee) {
    if (msg.sender == owner)
      fee = _fee;
  }
}
```

A sample smart contract (cont'd).

Fig. 3. Sample Contract.

money to the seller. Overall, due to the unpredictable selection of transactions in the pending pool (see Fig. 1), the persistent storage of the callee contract becomes "nondeterministic" from the perspective of end-users, and sample code in Fig. 3 has shown the feasibility of exploiting such nondeterminism to cause end-user's financial losses. This work aims to pinpoint contract payments that can be affected by malicious contract users, owners or even the underlying miners via the exploitation of nondeterminism. We start by discussing potential sources of nondeterminism in the following section.

2.3 A Nondeterministic Perspective of the Contract Execution Context

In this section, we present a nondeterministic perspective of the contract runtime system. We believe the lack of an in-depth understanding of such unique execution context leads to improper handling of various types of *nondeterminism* in Ethereum-involved scenarios. By formulating the contract execution context and the latent nondeterminism comprehensively, we point to the root causes of various common contract vulnerabilities (see Sec. 3).

Nondeterminism. It is worth noting that "nondeterminism" discussed in our research context is *highly inspired* by people's classical idea of how nondeterminism works in multi-threading programming. Indeed, existing research has shed light on the "Concurrent Perspective on Smart Contracts" [Sergey and Hobor 2017], where Ethereum transactions using the persistent storage of a contract (e.g., the `fee` variable in Fig. 3) is analogical to threads accessing concurrent objects in the shared memory. Overall, in Ethereum, the outcome of a transaction is *nondeterministic* from the perspective of the end-user because a lot can happen between when a transaction is dispatched and when it's incorporated in the blockchain. These unexpected behaviors are directly controlled by self-interested individuals in Ethereum. More specifically, malicious parties can manipulate the state of the contract to their benefit and cause a party to lose out. Miners can reorder transactions and set block properties (e.g., `Timestamp`; explained soon in Sec. 2.3.1) and other users can broadcast malicious transactions.

The present work takes a systematic and unified focus on the inherent nondeterminism of Ethereum (discussed in this section) and explain why they are the root causes of various common smart contract vulnerabilities (discussed in Sec. 3). As illustrated in Fig. 4, besides transaction scheduling, this research further extends and discusses two other key nondeterministic factors: external callee behaviors and Ethereum system properties.

2.3.1 Nondeterministic Block and Transaction States. As discussed in Sec. 2.1, a submitted transaction is usually retained in the pending pool for some time before being executed. The direct consequence of this delay is that the *block state* (e.g., the block number or timestamp) of the Ethereum blockchain becomes nondeterministic, since the expected value at the time of uploading the transaction could be drastically different from the obtained value during transaction execution.

Similarly, the *transaction state* property (`tx.origin`) is provided by the EVM for smart contracts to manage transaction logics and to check the identity of its caller. However, the frequently ignored fact is that `tx.origin` always refers to the contract originally initiating the transaction. Hence, a callee contract cannot verify the identity of its latest caller, given a chain of contract calls within a transaction. `tx.origin` is a source of nondeterminism in Ethereum, and as suggested [evm 2018b],

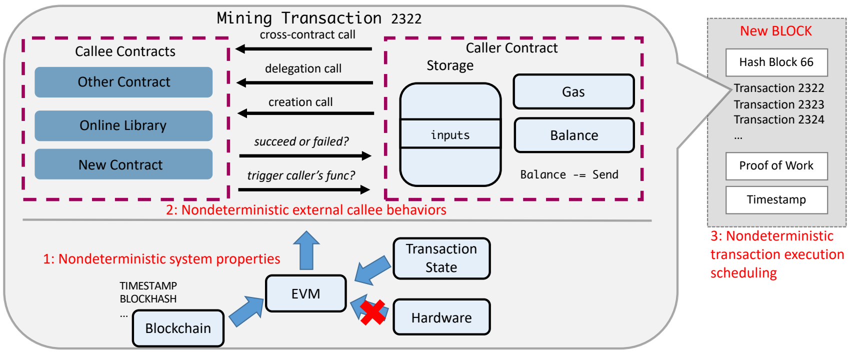

Fig. 4. A nondeterministic perspective of the smart contract execution context. We highlight three key nondeterministic factors that could introduce lurking nondeterministic payment bugs.

that contract developers should not rely on this value to guard any critical program logics such as issuing a payment. See Sec. 3.1 for working examples to exploit nondeterministic system properties.

2.3.2 Nondeterministic Transaction Execution Scheduling. As aforementioned, miners are free to pick transactions from the pending pool and determine the order of transaction execution. As a result, the assumptions concerning the program global variables of an online contract could be misleading due to the transaction execution races. Recall that, in general, each transaction dispatched to a smart contract invokes a function. Therefore, multiple transactions towards the same contract are comparable to threads accessing shared memory [Sergey and Hobor 2017]. Contract global variables, including all the data in the persistent storage region of a contract, could suffer from such read-write hazards. For instance, when the contract user of the `Intermediary` contract shown in Fig. 3 creates a buy transaction, what can really happen could be the following transaction execution sequence:

$$\texttt{setFee} \rightarrow \texttt{buy}$$

where the `setFee` transaction *front-runs* the buy transaction and changes the fee. Indeed, there are real-world attacks who affect the transaction execution order by squeezing in a transaction with a higher miner reward or by conspiring with the miner (see Sec. 3.4 for exploitations)

2.3.3 Nondeterministic External Callee. Another major issue is the unpredictable (malicious) behaviors of external callees. While a fundamental tenet of defensive programming suggests that we should not assume anything involving an external call outside of our code will work as claimed, most online smart contracts drift from this golden principle, presumably due to unawareness of the fact that a malicious callee can exploit the caller contract and thus control the transferred fund. A number of real-world attacks have been launched and led to severe financial losses.

As shown in Fig. 4, cross-contract function calls (referred to as "external calls" from this point onwards) lead to a new form of nondeterminism. Because external calls are synchronized, in order to transfer the execution flow, the miner will pause the execution of the current contract and switch to another contract. However, the behaviors of the callee contract are an obstacle to the caller: its *nondeterministic* actions could lead to financial losses by calling its caller and manipulating the caller's persistent storage. For example, suppose a user can withdraw money from his deposit in a Bank contract by calling the `withdraw` function of the contract. While this action seems normal, what can possibly happen is the following transaction execution sequence:

$$\texttt{withdraw} \rightarrow \texttt{withdraw} \rightarrow \texttt{withdraw} \rightarrow \ldots$$

where the malicious user keeps issuing new `withdraw` transactions to drain Ether from the Bank contract's balance when receiving the transferred money, and before the bank (which is *paused* and waiting for user's response) deducts the withdrawn amount from the user's deposit (see Sec. 3.3 for working examples of such attacks). Real world attacks have shown the feasibility to reenter such a payment function and drain all the balance of carelessly designed contracts [Siegel 2016].

In addition, external calls are designed in such a way that developers need to confirm manually that the operation indeed succeeds. In other words, the returned value of an external call is *nondeterministic* as well. The caller's execution will be reverted and all the funds are frozen, in case a failed external call is not properly handled. Case studies will be given shortly in Sec. 3.2.

3 NONDETERMINISTIC PAYMENT BUGS AS THE ROOT CAUSE OF COMMON CONTRACT VULNERABILITIES

In this study, we aim to capture unpredictable funds transfer due to the inherent nondeterminism of the Ethereum blockchain system. In the rest of this section, we will show that the awareness of nondeterminism (see Sec. 2.3) actually enables a more principled and unified way to describe a set of common vulnerabilities, such as the TOD and reentrancy bugs [evm 2018b; Siegel 2016]. For the ease of understanding, we will also discuss defenses programmers can employ to guard against these attacks. We will then scope the detectable vulnerabilities of NPChecker in Sec. 3.5, including many well-known issues as well as much subtler defects. Before introducing typical attacks, we first introduce the threat model of this research.

Threat Model. Some system-level design choices of the Ethereum blockchain lead to subtle nondeterminism in the contract execution context, which can cause logical errors between a contract's intended behavior and its implementation. In this research, attackers are assumed to be able to access the online smart contract and identify logical bugs of this kind. As a result, the attacker can deliberately exploit these logical bugs, by directly manipulating a malicious miner or using an adversarial contract to interact with the target contract. Attackers can also schedule front-running transactions toward the target contract to alert its intended behavior. Successful attacks will lead to undesired payments (i.e., "nondeterministic payment bugs") and cause potential financial losses for normal users.

3.1 System Property Dependence

As discussed in the best practices of Ethereum programming [evm 2018b], attackers can manipulate various system properties, including both block and transaction states, to exploit a contract. For instance, malicious miners can adjust block states such as the block timestamp and gas limit to manipulate the generation of randomness in a contract. Online contracts can tweak the transaction state properties (i.e., transaction origins) to hide its identity and bypass ownership checks of the target contract. Overall, all direct and indirect dependencies of the these *nondeterministic properties* should be taken care of cautiously, especially when funds transfer is involved. Consider a simple gambling game below:

```
contract RandomReward {
  uint256 constant private salt = block.timestamp;
  uint256 constant private threshold = 1000;

  function buggy_reward(uint256 bet) public {
      uint256 t = salt * block.timestamp/(salt % 5) ;
      if (t > threshold)
        msg.sender.send.value(bet * 100)().
  }
}
```

where the `block.timestamp` is used to compute a "random" number and further reward the message sender when the random number is larger than a threshold. While such a system state is fundamentally nondeterministic (as we have discussed in Sec. 2.3.1), using it as the seed for randomness entails *insecure* design because the block timestamp can be tweaked by miners, who can thus indirectly control the reward value and "beat the crowd."

Recommended Practice. Instead of depending on system properties which could be manipulatable, contracts can leverage the third party oracle service [ora 2019] that provides an audit trail to fetch random numbers from random.org. Besides, a decentralized scheme called RANDAO [ran 2019] is proposed to generate random numbers via crowdsourcing. For instance, each participant of a lottery-style gambling game generates her own random number offline and submit to a RANDAO contract. Then, the RANDAO contract performs user identity checks and computes the exclusive or of all submitted random numbers as the final result. While this scheme does not require a centralized oracle service that is "trustworthy", it imposes notable cost since extra transactions are needed to generate each random number.

3.2 Failed External Calls

As discussed in Sec. 2.3.3, external calls require developers themselves to take on responsibility for the success of operations. However, as suggested in the best practices of Ethereum programming [evm 2018b], improper handling of external call failure will likely result in exploitable contract behaviors, in case the failure is not isolated. Consider a simple crowdfunding contract below:

```
address[] private refundAddresses;
mapping (address => uint256) public refunds;
function refundAll() public {
  for(uint256 x; x < refundAddresses.length; x++) {
    // now a single failure on send will hold up all funds
    require(refundAddresses[x].send(refunds[refundAddresses[x]]))
  }
}
```

where the contract iterates through an array to refund the supporters of this crowdfunding project. The issue is that one supporter, when conspiring with the project coordinator, can deterministically cause this refund call to fail, for example, by crafting the fallback function of his contract intentionally runs out of gas. As a result, the refund loop can never complete and funds are locked. No one gets paid because of the *nondeterministic outcomes* of this external call and its non-isolated influence on further funds transfer.[2]

```
mapping (address => uint256) refunds;

function refundOne() external {
 uint256 refund = refunds[msg.sender];
 refunds[msg.sender] = 0;
 msg.sender.send(refund);
}
```

Recommended Practice. Intuitively, the contract can be amended such that each external call is isolated within a single transaction and therefore can minimize its (accidental or deliberate) damage. A commonly used design pattern is called "favor pull over push", where users will have to withdraw funds rather than wait for the the contract to push funds. Consider the revised crowdfunding contract on the left, where instead of sending refunds to all the crowdfunding supporters, the crowdfunding contract waits each user to claim the refund by calling `refundOne`. Each payment is handled within one transaction, which eliminates the possibility of locking funds.

3.3 Contract Reentrancy

Here we show that the well-known reentrancy bugs are actually due to the nondeterministic behaviors of the callee in a contract call. Consider a simple digital bank contract below:

```
contract Attack {
  function attack() { bank.withdraw(); }
  function () public payable { bank.withdraw(); }
}
contract Bank {
  mapping (address => uint256) private userBalances;
  function withdraw() public {
    uint256 amountToWithdraw = userBalances[msg.sender];
    msg.sender.call.value(amountToWithdraw)();
    // the attacker's code is executed, and calls withdraw again
    userBalances[msg.sender] = 0;
  }
}
```

[2]This bug is also referred as "non-isolated calls" or "unexpected revert" in some articles [evm 2018b; Grech et al. 2018].

where the contract Attack creates a transaction, and contract Bank, in turn, calls the `withdraw` function and sends attacker its deposit. This payment triggers the fallback function of Attack (the function with no name; when a contract issues a payment without specifying which function to call, the fallback function will be called by default) that repeatly invokes the `withdraw` function. However, because the balance of Bank has not yet been deducted, the second (and later) invocations will still succeed. With a deliberately crafted set of loops, attackers could deplete all the Ether in the digital bank. Due to the *nondeterministic behavior* of external callees, functions within the digital bank could be re-executed within a normal withdraw transaction, causing *read-write hazards* of contract Bank's global variables and further influence funds transfer.
Recommended Practice. As previously discussed (see Sec. 2.2), every EVM instruction execution costs certain amount of gas and in fact `call` will invoke the callee with almost all the gas the caller has. Therefore to fix the reentrancy issue, one common suggestion is to reduce the available gas the callee can use, by replacing `call` with two other external call methods `transfer` and `send`, which do not delegate enough gas for the malicious contract to change caller states.[3] Also, Ethereum recently supports a new opcode `STATICCALL`, which disallows modifications to caller states even if there are enough gas and prevents the reentrancy attacks in the first place.

3.4 Transaction-Ordering-Dependence

Due to the miner's *nondeterministic scheduling* of transactions (Sec. 2.3.2), program global variables are subject to read-write hazards as well. Consider the contract below:

```
contract Raffle {
    mapping(uint256 => address) reserved;
    function reserve(uint256 value) public payable {
    // check whether corresponding entry has been initialized or not
        if (reserved[value] == 0) {
          // can only enter once when uninitialized (0)
          reserved[value] = msg.sender;
        }
    }
}
```

where the buggy contract example presents an inconsistent program state affected by front running. This example acts as part of a simple raffle game, where users transfer certain amount of Ether to the payable function `reserve` and also reverse a number as the function input. Suppose a user (i.e., the "victim") aims to reserve a number 23, then anyone can see the scheduled transaction in the pending pool before it is committed. An attacker can schedule a transaction reserving the same number, and by raising the miner reward higher to the value of the victim's transaction, the malicious transaction will be executed *first*, initializing the corresponding entry in `reserved`. Hence, the victim's reverse will fail, leading to the loss of transaction fee and the paid Ether.
Recommended Practice. To defend TOD attacks, one common practice is to use a "pre-commit" scheme. Consider a revised raffle contract below (due to the limited space, this code is simplified):

```
contract Raffle {
 mapping(address =>bytes32) hashes;
 mapping(uint256 =>address) reserved;

 function commit(bytes32 hash){
  require(commits[msg.sender] == 0);
  hashes[msg.sender] = hash;
 }
```

```
 function reveal(uint256 v) public {
  bytes32 d = sha3((v, msg.sender));
  // verify the hash
  require(hashes[msg.sender] == d);
  require(reserved[v] == 0);
  reserved[v] = msg.sender;
 }
}
```

where the user first computes and sends a hash instead of his data by calling `commit`. Note that since `hashes` stores the submitted hash by user addresses, an attacker can send the same hash via `commit` but it will not prevent others from reserving the hash. Then, the victim reveals the data

[3] However, we note that the recent Constantinople upgrade of Ethereum enables low-cost state changes and therefore `transfer` and `send` can be vulnerable as well [ChainSecurity 2019].

that produces the committed hash by calling `reveal`, and update `reserved` with `v`. At this step, although the scheduled transaction (e.g., reserve a number 23 by calling `reveal`) can be seen by the malicious user from the pending pool, he is unable to exploit the `reveal` function because reverse engineering hash functions like `sha3` is very difficult.

3.5 Overview of Payment Bugs Detectable by NPChecker

We have discussed several common smart contract vulnerabilities and their root cause because of the inherent nondeterministic factors in the contract execution context. In this section, we introduce two kinds of payment bugs ($\boldsymbol{NP}_I$ and $\boldsymbol{NP}_{II}$) that are derived from the nondeterminism as follows:

- $\boldsymbol{NP}_I$: payment bugs belonging to this category describes contract *local variables* initialized by nondeterministic system properties or external call status. The affected local variables are checked on whether they can influence contract payments.
- $\boldsymbol{NP}_{II}$: payment bugs belonging to this category describes contract *global variables* under read-write hazards due to nondeterministic Ethereum transaction scheduling or interactions of other contracts. We check the affected global variables on whether they can influence contract payments.

Table 1. Classification of contract vulnerabilities. Note that "system properties" include usage of both nondeterministic block and transaction states.

	$\boldsymbol{NP}_I$ Bug	$\boldsymbol{NP}_{II}$ Bug
System property dependence	✓	
Failed external call	✓	
Reentrancy		✓
Transaction-order dependence (TOD)		✓

We summary vulnerabilities and their corresponding categories in Table 1. It is worth noting that each category of "vulnerability" can subsume *multiple variants*. For instance, reentrancy attacks include the aforementioned single-function reentrancy, as well as cross-function reentrancy which exploits inconsistent program states by reentering different functions of a contract to launch the attack. In addition, while the classic TOD attacks exploits a pair of transaction calls to attack a contract, recent studies also show the feasibility to exploit the so-called "event-ordering" (EO) bugs during which a contract can be exploited by deliberately crafting a long sequence of transaction calls [Kolluri et al. 2018]. The initiating cause of EO bugs is the nondeterministic scheduling as well. There could exist even tricker reentrancy attack vectors by crossing different contracts [Rodler et al. 2018]. For instance, EVM enables "contract delegation" for a contract to outsource its global storage and computations to some online library contracts with common reusable functionalities. While this feature allows more flexible development, it can also enable subtle reentrancy bugs, such that the delegatee library first calls an attacker contract, and the attacker contract further reentering the delegator contract (i.e., the "caller" of the library). In addition, a contract is allowed to create another contract, where the constructor function of the newly-created contract is called for initialization. However, a newly-created contract could perform (malicious) reentrancy by redirecting the execution flow from its constructor to functions of its creator [Rodler et al. 2018]. As studied in recent research, such cross-contract reentrancy attacks can impede existing static bug detectors, because it is not feasible to analyze individual contracts to reveal such hidden defects. However, shortly we will show that NPChecker can catch all such subtle issues.

3.6 Application Scope

The ability to directly analyze EVM bytecode (see the design and implementation sections in Sec. 5 and Sec. 9) implies a broader application scope of NPChecker compared to existing source code-based approaches [Kalra et al. 2018]. However, it is challenging to determine the intended behaviors of a contract because its true intention is generally obscure without public documents and with anonymous deployment. Hence, it is possible that some bugs reported by NPChecker are actually desired in the contract. Overall, we acknowledge the difficulty of "confirming" the existence of a bug.

Following the convention of previous studies, we aim to provide a tool that can assist developers or third-party analysts by flagging potential $\boldsymbol{NP}_I$ and $\boldsymbol{NP}_{II}$ defects. Users can then leverage the analysis results to minimize their manual inspection efforts. To emphasize the difference from previous studies which adopt relatively weak threat model by capturing vulnerabilities that lead to inconsistent program states, our threat model is more practical in the sense that we focus on funds transfer of nondeterministic amounts which presumably is undesired and indicate financial losses.

Our technique performs extensive modeling of nondeterministic factors in the Ethereum contract execution context which are the *root causes* of many common vulnerabilities (e.g., the reentrancy attack has recorded millions of dollars in losses [Siegel 2016]). Conceptually, we admit that there exist some "language-level" nondeterminism as well. For instance, the inputs of an online contract are unpredictable, and loops whose behavior is determined by user input could simply iterate too many times and become economically unrealistic (since every instruction of a contract costs certain amount of fee to execute). In this work, we do not consider such "language-level" uncertainty because we assume it is straightforward to provide sufficient validation checks to guarantee the legitimacy. Our major focus here is on more subtle system-level nondeterminism issues hidden in the contract execution context and runtime that are often ignored and mistaken by programmers due to the *unique execution model* of smart contracts. Also, deprecated/historical attacks (e.g., call depth attack [evm 2018b]) are also not considered, because they are no longer feasible due to improvements made in the protocol or Ethereum language.

3.7 Soundness and Completeness

This research aims to present a practical tool to detect bugs from online contracts in bytecode format. Our tool roots the same assumption with previous techniques [Luu et al. 2016; Nikolić et al. 2018; Rodler et al. 2018; Tsankov et al. 2018] that aim to *find bugs* rather than *rigorous verification*, which would come with its own set of challenges, such as the typical lack of nontrivial and complete contract-specific properties.

A strong distinction of our work from the previous "vulnerability checklist"-based bug detection is that our work is the first to provide a comprehensive summary of several nondeterministic factors in Ethereum and explain that they are the *root causes* of many existing attacks. In contrast, the "vulnerability pattern matching"-based techniques require a constantly-updated check list, which is likely to be defeated by new vulnerabilities or variants of known issues, as we will show in Sec. 10.1.

Soundness. As discussed in Sec. 3.6, we acknowledge that there exist certain "language-level" uncertainties such as integer overflow/underflow issues that our work does not consider (existing research has proposed techniques to pinpoint such issues [Grech et al. 2018]). In contrast, we propose a principled modeling of subtle system-level nondeterminism, and perform information flow tracking to reveal the influence of nondeterminism on contract payments. Conceptually, we have modeled all the system-level nondeterminism, and our information flow tracking shall expose any payments that are potentially affected by these nondeterministic factors. In this sense, our technique is *sound* since we model the "upper bound" of potentially undesired payment influenced by the analyzed nondeterminism.

From an implementation perspective, our prototype is not sound. Inspired by how a multi-threaded program is instrumented into a sequential representation for model checking [Qadeer and Wu 2004], we model nondeterministic transaction scheduling with a sequential representation where the instrumented program simulates a practical subset of possible contract executions. Besides, our model checker performs bounded model checking (see Sec. 9), meaning that in principle we may miss some findings.

Completeness. From the *design* perspective, our threat model assumes, in general, that any funds transfer influenced by nondeterminism is undesired and presumably "vulnerable." However, tentative studies show that there exist certain cases where this is actually the intended behavior, i.e., there are legitimate situations where "nondeterministic" behavior seems intentionally used to influence payments. We will present relevant cases in Sec. 10.3 and Sec. 11 to enable a more thorough understanding. Overall, we acknowledge the difficulties in comprehending the intended behavior of a contract (see Sec. 3.6), since smart contracts are usually deployed anonymously in its bytecode format without any public documentations. Therefore, our analysis could lead to false positives (i.e., incomplete). The main users of our work are contract developers or third-party security analysts — they can use our technique to "debug" contracts and fix suspicious program locations that may be exploited.

Regarding the *implementation* of our prototype NPChecker, we found several decompilation errors for the employed decompiler which largely inflate the decompiled programs with irrelevant code fragments. This would led to some false positives in our findings as well (discussed in Sec. 10). Also, since we do not differentiate `STATICCALL` and `CALL` in our prototype implementation (see Sec. 6), this may also lead to false positives.

4 RELATED WORK

We now review existing work in detecting vulnerabilities in smart contracts. Symbolic execution has been widely used to analyze program paths and capture some predefined vulnerability patterns [ConsenSys 2018; Feng et al. 2019; Luu et al. 2016]. We will compare our work with Oyente and Mythril's and demonstrate how NPChecker catches much more defects. Securify [Tsankov et al. 2018] extracts Datalog facts from EVM bytecode and perform sound analysis to decide the presence of bugs regarding predefined patterns. FSolidM [Mavridou and Laszka 2018] allows users to define contracts as finite state machines, thus enabling the verification of contract semantics and eliminating several common vulnerabilities in the first place. MAIAN [Nikolić et al. 2018] uses symbolic analysis to flag specific vulnerable code patterns that can lead to funds locking indefinitely or being drained by arbitrary users. ZEUS [Kalra et al. 2018] performs sound abstract interpretation to verify the correctness of contracts with respect to several bug patterns. ContractFuzzer [Jiang et al. 2018] utilizes fuzz testing on smart contracts, with testing oracles defined for several known issues. Some other research, including MADMAX, which focuses on a set of gas-focused vulnerabilities associated with denial-of-service attack [Grech et al. 2018], and TeEther [Krupp and Rossow 2018], which synthesizes exploitation towards online contracts, is orthogonal to our work. In addition, some runtime monitoring techniques enforce data-flow integrity to defeat reentrancy attacks [Rodler et al. 2018]. We also notice a line of research work verifying smart contracts with formal methods [Amani et al. 2018; Bhargavan et al. 2016; Grishchenko et al. 2018; Hildenbrandt et al. 2018; Hirai 2017]. However, this kind of approach usually requires the interaction with human when proving, and also focuses on functionality specifications instead of security properties.

Some recent studies laid a good foundation on revealing the concurrent execution model of the Ethereum blockchain system [Kolluri et al. 2018; Sergey and Hobor 2017]. While their analysis of transaction interference is more comprehensive, a systematic modeling of nondeterminism that could induce various attacks is still absent. Moreover, their research over aggressively treated any program states that can be changed by reordering transactions as "buggy." We take a step back in this regard and perform a more practical analysis by pinpointing transferred funds that can be affected nondeterministically.

To demonstrate the expressiveness of our tool, we now give an overview of the existing tools in this field ordered by the publication date, focusing on common vulnerabilities found in research and public media. Table 2 presents the summary of existing research. In contrast to the pattern matching-based approach, the principled modeling of Ethereum nondeterminism in NPChecker naturally captures a comprehensive set of common vulnerabilities. More importantly, NPChecker is much more inclusive in terms of new issues (see Sec. 10.1 for empirical studies of new issues). In contrast, most of existing works are based on predefined bug patterns and therefore is difficult to direct flag new variants of known vulnerabilities (marked with *diamond* in Table 2). It is worth mentioning that although the current release does not support to find dependencies on system properties [sec 2019], Securify provides a DSL that users can use to specify new patterns (e.g.,

Table 2. A comparison of existing smart contract bug detectors w.r.t. the expressiveness of common vulnerabilities. ◇ represents limited support. Sereum [Rodler et al. 2018] instruments the EVM to flag suspicious online transactions while the rests perform static analysis.

Tool	**General to Flag New Bugs**	**Common Vulnerabilities**					
		Reentrancy	System Properties	TOD	Failed External Calls	Unbounded Loop Operations	Arithmetic
OYENTE [Luu et al. 2016]		◇	◇	◇			
Mythril [ConsenSys 2018]		◇	◇				✓
Zeus [Kalra et al. 2018]		◇	✓	◇	✓		✓
Sereum [Rodler et al. 2018]	✓	✓					
MADMAX [Grech et al. 2018]					✓	✓	◇
ContractFuzzer [Jiang et al. 2018]		◇	✓	◇			
Securify [Tsankov et al. 2018]		◇		◇			
EthRacer [Kolluri et al. 2018]	◇			✓			
NPChecker	✓	✓	✓	✓	✓		

dependencies on `timestamp`). Nevertheless, their flexibility to flag new bugs depends on new patterns proposed by experienced security analysts, and presumably new patterns would not be created, unless certain real-world attacks have occurred to draw the community's attention. Also, not all the properties analyzed by NPChecker can be encoded into their DSL, for instance, to analyze cross-contract reentrancy attacks, as discussed in Sec. 3.5, two contracts need to be taken into consideration simultaneously.

Similarly, to detect the Transaction-Ordering Dependency (TOD) bug, current studies mostly place emphasis on detecting program states subject to read-write hazards by reordering the execution of two functions. In contrast, our work (and also another recent work [Kolluri et al. 2018]) unearths a much larger pool of attack possibilities by systematically modeling the interference of an arbitrary number of functions.

Yet there are some vulnerabilities that NPChecker does not support well. For instance, NPChecker is not designed to symbolically interpret the program execution. Therefore, arithmetic issues such as overflow are beyond the scope of this work. We refer readers to some orthogonal research which analyzes such issues [Grech et al. 2018].

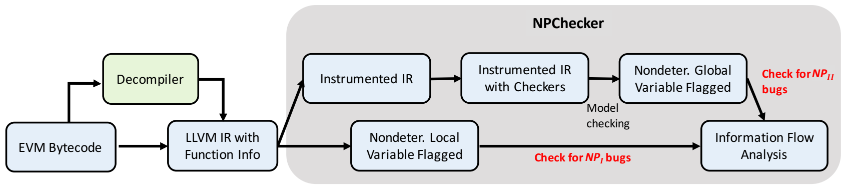
NPChecker
Decompiler
EVM Bytecode
LLVM IR with Function Info
Instrumented IR
Instrumented IR with Checkers
Nondeter. Global Variable Flagged
Check for NP_{II} bugs
Model checking
Nondeter. Local Variable Flagged
Check for NP_I bugs
Information Flow Analysis

Fig. 5. Workflow of NPChecker.

5 DESIGN OF NPCHECKER

We now present NPChecker, a tool that uses static analysis to detect payment bugs in smart contracts. Fig. 5 summarizes the workflow of NPChecker. Given the EVM binary code collected from the Ethereum mainnet, we first disassemble the EVM bytecode and lift the assembly instructions to LLVM IR using an off-the-shelf LLVM IR lifter (EVMJIT [evm 2018a]). In short, each EVM instruction is translated into one or multiple lines of LLVM IR statements. All the EVM instructions that are specific to smart contracts, such as external contract calls, get translated into LLVM external calls. The salient features of the LLVM IR code for smart contracts will be given in Sec. 6. In addition, NPChecker recovers the control flow structures with the help of a commercial decompiler. Given this retrieved high-level program information, we augment the LLVM IR with functions. The technical details concerning the reverse engineering process are discussed in Sec. 9.

We then seek to check whether the recovered IR code contains $\boldsymbol{NP_I}$ and $\boldsymbol{NP_{II}}$. As shown in Fig. 5, to check the existence of $\boldsymbol{NP_I}$, we identify local variables that are initialized with nondeterministic values (i.e., system properties and external call status). We then perform information flow tracking

Algorithm 1: Detecting nondeterministic payment bugs. Line 1–11 detects $\boldsymbol{NP_I}$ bugs and line 12–32 detects $\boldsymbol{NP_{II}}$ bugs.

```
Input: LLVM IR program $p$ lifted from EVM bytecode
Output: (Has $NP_I$ bugs?, Has $NP_{II}$ bugs?)
1  $\mathcal{V} \leftarrow$ nondeter_taint_source($p$); /* flag local variables initialized with nondeterministic values. */
2  $\mathcal{F} \leftarrow$ fund_transfer($p$); /* flag all the fund transfer statements. */
3  $Vul_I \leftarrow$ false;
4  $Vul_{II} \leftarrow$ false;
5  for each $v$ in $\mathcal{V}$ do
6      taint_propagation($v$);
7      if check_taint_sink($\mathcal{F}$) then
8          $Vul_I \leftarrow$ true; /* if any funds transfer depends on $v$. */
9          break;
10     end
11 end
12 $p' \leftarrow$ instrumentation($p$); /* instrumentation to simulate the potential runtime behaviors of a contract. */
13 $p'' \leftarrow$ insert_checkers($p'$); /* insert checkers to detect read-write hazards. */
14 $execution\_time \leftarrow 0$; /* check for timeout */
15 $\mathcal{S} \leftarrow \varnothing$;
16 while $execution\_time <$ THRESHOLD do
17     /* model checking to find read-write hazards of a global variable; variables in $\mathcal{S}$ are excluded. */
18     $v \leftarrow$ find_counterexample($p''$, $\mathcal{S}$);
19     if $v == null$ then
20         /* no counterexample indicates the safety of the checked program. */
21         $Vul_{II} \leftarrow$ false;
22         break;
23     end
24     add $v$ in $\mathcal{S}$;
25     taint_propagation($v$);
26     if check_taint_sink($\mathcal{F}$) then
27         $Vul_{II} \leftarrow$ true; /* if any funds transfer depends on $v$. */
28         break;
29     end
30     update_executed_time($execution\_time$);
31 end
32 return ($Vul_I$, $Vul_{II}$);
```

to analyze their influences on the funds transfer (i.e., detecting $\boldsymbol{NP_I}$ bugs). To check the existence of $\boldsymbol{NP_{II}}$, we instrument and augment the IR code with extra components to model the contract execution context. We further insert checkers and employ a model checking engine to flag program global variables (i.e., data in persistent storage) under read-write hazards. The pinpointed program variables are deemed to have nondeterministic values and they are as well fed to the information flow tracking module to study their influences on payments.

Alg. 1 specifies the detailed workflow of NPChecker (the grey box in Fig. 5). Given an LLVM IR program p lifted from a piece of EVM bytecode, we first collect all the local variables that are initialized with nondeterministic values (line 1) and all the funds transfer statements (line 2). Taint analysis (line 5–11) serves to check if any payments depend on the system properties or external call returns. Dependencies found at this step constitute $\boldsymbol{NP_I}$. We further check bugs belonging to $\boldsymbol{NP_{II}}$ by identifying funds transfer that depends on nondeterministic program global variables. To do so, we instrument the IR code by modeling nondeterministic transaction scheduling and external callee behaviors. As a result, we practically simulate a large subset of the contract execution behaviors (line 12). We further insert checkers (line 13) and leverage model checking techniques to detect global variables suffering from read-write hazards (line 18–23). After flagging program global variables whose value are nondeterministic, we perform taint analysis (line 25–29) with the flagged nondeterministic global variables as the taint source, and the payment statements (see Sec. 6 for the definition of "payment statements") as the taint sink.

Our taint analysis module keeps track of all the tainted variables (with a LLVM `DenseSet`) and their propagations. `taint_propagation` inserts a new element in the set and starts to propagate

$$\begin{array}{lll}
\text{Property} & pty & ::= \texttt{GasPrice} \mid \texttt{TxOrigin} \mid \texttt{Miner} \mid \texttt{BlockNum} \mid \texttt{TimeStamp} \\
 & & \mid \texttt{GasLimit} \mid \texttt{Difficulty} \\
\text{TxID} & tid & \in \mathbb{N} \\
\text{Products} & prods & ::= \texttt{define}\ f(tid)\{c\} \mid \texttt{declare}\ f(tid) \mid \texttt{global}\ v \\
\text{Address} & addr & ::= addr_0 \mid addr_1 \mid addr_2 \mid \ldots \\
\text{Ops} & \otimes & ::= \texttt{add} \mid \texttt{sub} \mid \texttt{mul} \mid \texttt{udiv} \mid \texttt{sdiv} \mid \texttt{shl} \mid \texttt{lshr} \mid \texttt{and} \mid \texttt{or} \mid \texttt{xor} \mid \ldots \\
\text{Variable} & v & ::= v_0 \mid v_1 \mid \ldots \mid l \\
\text{Expression} & e & ::= n \mid v \mid e_1 \otimes e_2 \mid \neg e \\
\text{Command} & c & ::= v := e \mid c_1; c_2 \mid v_1 = \texttt{load}(v_2) \mid \texttt{store}(v_1, v_2) \mid v_3 = \texttt{icmp}(cond, v_1, v_2) \\
 & & \mid \texttt{ret}(v) \mid v_2 = \texttt{malloc}(v_1) \mid \texttt{free}(v) \mid v_2 = \texttt{alloca}(v_1) \mid \texttt{br}(v, c_1, c_2) \\
 & & \mid v_1 = \texttt{sload}(v_2) \mid \texttt{sstore}(v_1, v_2) \mid \texttt{call}(addr, v_1, v_2) \mid v_2 = \texttt{delegate}(addr, v_1) \\
 & & \mid \texttt{create}(addr, v_1, v_2) \mid \texttt{destruct}(addr) \mid pty = \texttt{blockchain}(v_1)
\end{array}$$

Fig. 6. Syntax of LLVM IR lifted from smart contracts. Note that we only specify the core components in a terse way to present the main idea behind NPChecker. We simply omit some LLVM syntax definitions (e.g., `phi` node) as well as some EVM components for the sake of space and readability. We highlight EVM specific code and properties with blue and purple respectively.

its information flow following the pre-defined taint policy (see Sec. 8). `check_taint_sink` checks each external call and sees whether the amount of any payment is tainted or not. As emphasized, the "tainted" funds transfer is presumably spurious, indicating potential financial losses.

In the following sections, we start by formulating the recovered LLVM IR in Sec. 6. In Sec. 7, we introduce how the LLVM IR code is instrumented to expose contract global variables under read-write hazards (line 12-13 in Alg. 1). We describe information flow tracking in Sec. 8.

6 LANGUAGE DEFINITION

This section introduces a succinct IR language to demonstrate the core components of a smart contract. Sec. 7 and Sec. 8 further specifies our instrumentation and analysis rules with respect to the defined language. NPChecker is capable of handling LLVM IR code lifted from real-world smart contracts. To concisely present the main ideas behind NPChecker, we formalize its analysis to a *simplified representation* that is capable of modeling the essential parts of LLVM IR code lifted from EVM bytecode. Fig. 6 shows the syntax of LLVM IR language for Ethereum. We primarily extend a conventional LLVM IR core with various EVM and smart contract specific components.

In general, each LLVM IR file is a module that includes data, an information layout (omitted in our definition), and a list of *prods* $\in$ Products that can be function declarations, function definitions, or global variables. In the context of our research, functions in Products represent public methods defined in each smart contract, including the anonymous "fallback" function. In our definition, each function takes a transaction id $tid \in \mathbb{N}$ as input. We note that although in reality, each contract function actually takes some other inputs containing the caller information (e.g., function call inputs), they are omitted because they do not affect the presentation of our key idea.

Address denotes a unique 128-bit contract address on the Ethereum mainnet. OPs are standard LLVM IR operators, and Expression stands for an expression. The essential features of the LLVM instruction set are recorded in Command, such as assignment ($v := e$), sequential composition ($c_1; c_2$), block terminators (`br` and `ret`), heap operations (`malloc` and `free`), stack allocation (`alloca`), and comparisons (`icmp`). For EVM specific commands, `call` (i.e., the *payment* statement) transfers a certain amount of Ether v_1 to a recipient (callee) with the address *addr* and the function call input (if exists) v_2.[4] `destruct` terminates the current contract and transfers its balance to a given address *addr* (usually its owner). `delegate` performs delegation calls and dispatch the caller's program state

[4]Note that after lifting EVM bytecode into LLVM IR (see Sec. 9), payment related opcodes (`CALL`, `STATICCALL`, `DELEGATECALL`, `CREATE`, `CALLCODE`) are subsumed into a unified interface and differentiated by a parameter of this interface. In our prototype implementation, we instrument this unified interface, and therefore, all these opcodes are taken into account. For ease of presentation, we use `call` to subsume `CALL`, `STATICCALL`, and `CALLCODE` in the rest of the paper.

to an library smart contract with v_1 as the input and *addr* as the library address. Additionally, `create` creates another contract online with *addr* as the contract address, v_1 as the transferred Ether, and v_2 as the input value for the new contract's construction function. The contract global memory region, storage, are accessed via `sload` and `sstore`. v_2 and v_1 represent the index of the accessed global variable in `sload` and `sstore`, respectively. In addition, we create an extra global variable l indicating the aliveness of a contract. The default value of l is true for every online contract, but l will be set as false after the contract is self-destructed (by calling EVM `destruct` instruction). Note that by explicitly representing the contract aliveness via a global variable, subtle transaction races between normal transactions and "suicide" transactions can be modeled as well (see Sec. 7.3.2 for the implementation).

Table 3. The system properties accessible by a contract during runtime.

Index	Notation	Explanation
0	`GasPrice`	The transaction gas price.
1	`TxOrigin`	The transaction origin account.
2	`Coinbase`	The the beneficiary address of the block.
3	`BlockNum`	The block number.
4	`Timestamp`	The block timestamp.
5	`GasLimit`	The block gas limit.
6	`Difficulty`	The block difficulty.

As discussed in Sec. 2.3.1, the nondeterministic system properties are provided by the blockchain system. Here we use Property to denote information accessible by smart contracts when a transaction is being executed, including a comprehensive set of transaction state and block states. We further define `blockchain` : Integer → Property as a partial function that maps an integer to one system property (recall "system properties" subsume block and transaction states in Sec. 3.1) that can be accessed during transaction execution. The accessible runtime properties by a contract are specified in Table 3, including its corresponding index and explanations.

7 EXPOSE NONDETERMINISM WITH INSTRUMENTATION

In this section, we perform systematic instrumentation for the IR defined in Sec. 6. We aim to expose various nondeterministic program global variables hidden in the IR.

7.1 Modeling Transaction Scheduling

We start by defining a `mining` function to describe how transactions within one block will be scheduled for computation. Given a nondeterministic `choice` statement that chooses among different functions and an iteration statement `iter`$\{s\}$ that executes an unbounded number of times, we formulate the miner decision into a `mining` function as follows:

$$\texttt{mining}() = \texttt{iter}\{\texttt{choice}\{f_0 \| f_1 \| \ldots \| f_n\}\}$$

Note that exactly one function is executed nondeterministically in each iteration. This function simulates the behavior of a miner selecting transactions from the pending pool and executes each transaction by following certain mining strategies defined in `choice`. For instance, one common approach [Geth 2018] is to sort transactions by the gas price they are willing to pay.

Implementation of the `mining` Function. We now propose an implementation to approximate the nondeterministic `mining` function defined above. Enlightened by how multithreading programs are instrumented into a sequential representation for model checking [Qadeer and Wu 2004], we transform the `mining` function into a representation that practically simulates a large subset of the behaviors of contract execution.

The `schedule` function in Fig. 7 helps model the transaction scheduling policy of the miner (i.e., the `mining` function). `$` is a nondeterministic value that leads to unpredictable iterations of the `while` loop. The variable `tid` as the transaction id is used to distinguish function execution in different transactions. The `schedule` function simulates an arbitrary number of method calls to a particular smart contract, and function `get` randomly picks a function from all the public functions defined in that contract. Note that each entry in `fl` maintains a tuple of a function pointer and a counter (initialized as N) to track the number of function invocations within a contract. That is, per definition we allow the execution of each function at most N times.

```
// transaction ID
var tid = 1;
schedule() {
  while ($) {
  // f is a method reference
       var f = get();
       f(tid);
       tid++;
  }
}
```

The `schedule` function.

```
get() {
  var idx = $ % len(fl);
 // fl: keeps track of all the public
 // functions of a particular contract
  var t = fl[idx];
 // each fl.counter is initialized as N
  if (fl[idx].counter == 0)
      delete fl[idx];
  else
      fl[idx].counter -= 1;
  return t.fp;
}
```

The `get` function.

Fig. 7. Implementation of the `mining` function.

In the implementation, N is set to 2. This is empirically decided to reduce the analysis overhead by allowing the model checker to re-execute each function at most twice. Overall, the `schedule` function, which encapsulates the scheduling policy of the miner and interprets all functions for nondeterministic iterations, explores all the possible combinations of functions, while each function is used at most twice. The current modeling sufficiently explores practical attack vectors including single-function and cross-function reentrancy, while neglects cases where a function is invoked for more than twice (but unlikely to unveil new issues). In addition, while this implementation assumes a completely nondeterministic scheduler, a more sophisticated strategy could be provided by changing `get`, for instance, one that prioritizes transactions with higher miner rewards.

7.2 Instrumentation

We now specify a set of instrumentation rules for the IR language defined in Sec. 6. The instrumented program forms a practical basis for rigorous formal analysis and vulnerabilities detection.

$$\begin{array}{lcl}
(v = \texttt{call}(a, v_1, v_2))^{\mathcal{I}} & \rightarrow & \texttt{schedule}(); v = \texttt{call}(a, v_1, v_2) \\
(v = \texttt{delegate}(a, v_1))^{\mathcal{I}} & \rightarrow & \texttt{merge}(a); \texttt{schedule}(); v = \texttt{delegate}(a, v_1) \\
(v = \texttt{create}(a, v_1, v_2))^{\mathcal{I}} & \rightarrow & \texttt{schedule}(); v = \texttt{create}(a, v_1, v_2) \\
(\texttt{c})^{\mathcal{I}} & \rightarrow & \texttt{c} \ \text{otherwise}
\end{array}$$

Fig. 8. Definition of instrumentation function $\mathcal{I}$.

Given the definitions of functions `schedule` and `get` to approximate transaction scheduling of a miner, Fig. 8 illustrates our instrumentation rules with respect to the IR program defined in Fig. 6. To model nondeterminism in external calls and to capture the underlying issues (e.g., reentrancy attacks), we insert the `schedule` function before every `call` to simulate the nondeterministic behaviors of a callee contract. Since the `schedule` function practically models the execution of every public function within the contract (including the caller function f_i), the consequence is that we systematically explore the situations to *reenter* f_i and other functions within the same contract.

By providing a utility function `merge` which merges functions of the caller contract and its callee (i.e., an online library) together, we take the library contract into account and track down subtle cross-library issues (see Sec. 9 for the implementation of `merge`). In addition, `create` is instrumented to invoke the `schedule` function, which indicates the nondeterministic behaviors of the newly-created contract and the potential attack vector (e.g., creation-based reentrancy attacks).

As aforementioned, every public function of a smart contract could serve as the entry point of a transaction. In contrast, at this step we extend the instrumented code with the following `main` function as a unified entry point of a contract program.

$$\texttt{main}() = \{schedule();\}$$

For every contract function f_i, `schedule` can model an arbitrary amount of "front-running" transactions in front f_i. As we introduced in Sec. 3.4, the order of transactions is prone to manipulation within one block on the blockchain. In addition, it is easy to see that `main` function translates the contract program into a nondeterministic albeit sequential representation, which forms the basis to be checked by any sequential model checkers.

7.3 Expose Program Global Variables of Nondeterministic Values

As aforementioned, the major challenge is to expose program global variables that suffer from both read/write and write/write hazards due to unpredictable transaction scheduling and external callee behaviors. Enlightened by conventional research in detecting race conditions, we use model checking to discover global variables whose values are nondeterministic. The flagged global variables at this step will be sent to the information flow module to check $\boldsymbol{NP}_{II}$.

7.3.1 Implementing Checkers. We first define a set of auxiliary variables that act as indicators of accesses to program global variables. Specifically, we define $\texttt{access}_i \in \mathbb{Z}$ where $\texttt{access}_i = 1$ indicates that a read access has occurred. For a write access from a transaction with id tid, we will update $\texttt{access}_i$ with $tid + 1$. Each $\texttt{access}_i$ is initialized to 0. We now introduce a pair of checkers to find program states suffering from read-write hazards.

```
check_w(g_i, tid) {
 access_i = get_auxiliary_var(g_i);
 assert(!(access_i > 1)
       || access_i == tid+1);
 access_i = tid+1;
}
```

The $\texttt{check}_w$ function.

```
check_r(g_i, tid) {
 access_i = get_auxiliary_var(g_i);
 assert(!(access_i > 1)
       || access_i == tid+1);
 access_i = 1;
}
```

The $\texttt{check}_r$ function.

Function $\texttt{check}_w$ takes a global variable g_i and a transaction id tid as input, and then computes the index i of g_i and fetches the corresponding variable $\texttt{access}_i$ (see Sec. 9 for the implementation of `get_auxiliary_var`). Before writing to a global variable g_i, function $\texttt{check}_w$ check whether g_i has been written or read before. When the assertion succeeds, $\texttt{access}_i$ is set to $tid + 1$ to indicate that a write access has occurred within transaction tid. Similar to $\texttt{check}_w$, $\texttt{check}_r$ checks for the occurrence of write accesses before reading g_i. The corresponding $\texttt{access}_i$ variable will be set to one whenever the assertion succeeds. It is easy to see that an assertion in one of these calls is violated only if there are conflicting read/write accesses from two different contract function calls invoked by `schedule`. That is, we allow read accesses to g_i simultaneously within multiple transactions, while write accesses are allowed only within the same transaction.

7.3.2 Inserting Checkers. Given the checkers for read and write accesses, Fig. 9 specifies the statements which we insert the checkers to. In general, we insert checkers to flag program global variables (including global variables and the ghost variable l representing the contract aliveness) that can be affected by nondeterministic contract executions. In the given example, we show how we do this for `sload` and `sstore` statements. To capture the potential self-destruction races of two transactions, we place checkers in front of each payment statement and `destruct` statement to check read-write hazards on l. The `destruct` function will cause a write access to the ghost aliveness variable.

$$
\begin{array}{lll}
(v = \texttt{sload}(v_1))^{\mathcal{A}} & \rightarrow & \texttt{check}_r(v_1, tid); v = \texttt{sload}(v_1) \\
(\texttt{sstore}(v_1, v_2))^{\mathcal{A}} & \rightarrow & \texttt{check}_w(v_1, tid); \texttt{sstore}(v_1, v_2) \\
(v = \texttt{call}(a, v_1, v_2))^{\mathcal{I}} & \rightarrow & \texttt{check}_r(l, tid); v = \texttt{call}(a, v_1, v_2) \\
(v = \texttt{create}(a, v_1, v_2))^{\mathcal{I}} & \rightarrow & \texttt{check}_r(l, tid); v = \texttt{create}(a, v_1, v_2) \\
(\texttt{destruct}(addr))^{\mathcal{A}} & \rightarrow & \texttt{check}_w(l, tid); \texttt{destruct}(addr) \\
(c)^{\mathcal{A}} & \rightarrow & \text{c otherwise}
\end{array}
$$

Fig. 9. Definition of $\mathcal{A}$: augmented instrumented IR code with assertions. Note that the program global storage is accessed via `sload` and `sstore`, and therefore we capture the read-write hazards by instrumenting all the occurrence of these two `sload` and `sstore` statements. l represents the aliveness of the contract, and we add its corresponding assertions in front of payment statements and the `destruct` statement.

7.3.3 Model Checking. With the instrumented program we leverage an off-the-shelf model checker to identify program global variables that may be affected by nondeterminism (we use Smack [Rakamarić and Emmi 2014]; see Sec. 9 for implementation details). Flagged global variables (except the contract aliveness variable l) are sent to the information flow tracking module and check their influences on contract funds transfer. In case l is flagged as nondeterministic, the contract is deemed vulnerable regarding $\boldsymbol{NP}_{II}$ without information flow checking, because a suicided contract trivially affect the funds transfer.

8 CAPTURING NONDETERMINISTIC PAYMENT BUGS WITH INFORMATION FLOW TRACKING

We now identify spurious funds transfer liable to nondeterminism. As defined in Alg. 1, the taint analysis module serves to analyze information flow of local variables initialized with nondeterministic values and global variables under read-write hazards. Payments that depend on these two types of nondeterministic variables constitute $\boldsymbol{NP}_I$ and $\boldsymbol{NP}_{II}$, respectively.

8.1 Taint Checking

We first enumerate the policies for information flow checking defined in Fig. 10. To pinpoint nondeterministic system properties that can lead to $\boldsymbol{NP}_I$, we taint the local variable holding the output of the `blockchain` function (recall `blockchain` subsumes the block and system states accessible during a transaction). In addition, we taint the local variable holding the return value of `call`, `create`, and `delegate` statements to model the nondeterministic external call status. When checking $\boldsymbol{NP}_{II}$, we use the counterexmaples generated by the model checker to flag program global variables under read-write hazards. Such global variables will be tainted when they are accessed by `sload` and `sstore` statements.

We use `is_tainted` to check the amount of Ether (denoted by v_1) sent out by `call` and `create`: a tainted v_1 indicates that the amount of a payment is nondeterministic. Such nondeterminism is presumptively unwanted and deceptive.

Our taint analysis module is implemented as an LLVM pass and maintains an LLVM `DenseSet` to keep track of all the tainted variables. `taint` function extends the dense set by inserting a new element. Similarly, `is_tainted` checks the existence of a variable in the dense set, which indicates this variable is "tainted."

$$
\begin{array}{lll}
(v = \texttt{blockchain}(v_1))^{\mathcal{T}} & \rightarrow & v = \texttt{blockchain}(v_1); \texttt{taint}(v) \\
(v = \texttt{call}(a, v_1, v_2))^{\mathcal{T}} & \rightarrow & \texttt{is\_tainted}(v_1); v = \texttt{call}(a, v_1, v_2); \texttt{taint}(v) \\
(v = \texttt{delegate}(a, v_1))^{\mathcal{T}} & \rightarrow & v = \texttt{delegate}(a, v_1); \texttt{taint}(v) \\
(v = \texttt{create}(a, v_1, v_2))^{\mathcal{T}} & \rightarrow & \texttt{is\_tainted}(v_1); v = \texttt{create}(a, v_1, v_2); \texttt{taint}(v) \\
(v_2 = \texttt{sload}(v_1))^{\mathcal{T}} & \rightarrow & v_2 = \texttt{sload}(v_1); \texttt{taint}(v_2) \\
(\texttt{sstore}(v_1, v_2))^{\mathcal{T}} & \rightarrow & \texttt{taint}(v_2); \texttt{sstore}(v_1, v_2) \\
(\texttt{c})^{\mathcal{T}} & \rightarrow & \texttt{c} \ \ \text{otherwise}
\end{array}
$$

Fig. 10. Definition of $\mathcal{T}$: augmented IR code with information flow tainting and checking routines. Tainting of `sload` and `sstore` are performed only if the accessed global variable (v_1 is the global variable index) is flagged as under read-write hazards by the model checker.

8.2 Taint Propagation Policy

The taint propagation policies in our study follow the convention to model both *explicit* and *implicit* information flows. Explicit information flow is modeled in a straightforward way: we propagate the variable-level (i.e., registers and memory cells) information flow within a contract function. We also model the implicit information flow such that all the accessed variables will be tainted in a LLVM code branch, in case its guarded condition depend on the taint source. In addition, when the memory is accessed via a tainted memory address (either base address or the index), NPChecker taints the accessed memory content, indicating the information propagation from the tainted address to the memory cell. While our taint analysis is essentially intra-procedural, when encountering the usage of some LLVM or EVM auxiliary functions (e.g., the LLVM intrinsics functions), we will conservatively taint the function return value, whenever any of its parameters is tainted. As for external calls, the taint propagation will follow the definitions presented in Fig. 10.

9 IMPLEMENTATION

Most of the online contracts provide only EVM bytecode. Therefore, we implement NPChecker to directly process EVM bytecode and capture contract vulnerabilities in a "down-to-earth" manner. Different contract languages (e.g., Solidity and Vyper) are supported by NPChecker as long as they can be compiled into EVM bytecode. Contracts with source code available are of course analyzable once they have been compiled into bytecode.

NPChecker is written primarily in Python and C++ in about 4,800 lines of code. We lift the EVM bytecode of a contract into LLVM IR code with the off-the-shelf JIT compiler (EVMJIT [evm 2018a]) and further build our analysis framework on top of LLVM IR. EVM bytecode does not contain any function information. After being lifted into LLVM IR, the EVM bytecode forms one monolithic LLVM IR code block. Therefore, we recover information of function starts from the EVM bytecode (including the anonymous "fallback" function) using a commercial decompiler, JEB3 [PNF 2018]. The acquired function starts are then used to traverse the intra-procedural CFG of the LLVM IR code and split basic blocks within the monolithic code block into functions.

We then perform instrumentation towards LLVM IR with the aforementioned techniques. We also implement our taint analysis module as one LLVM IR pass to keep track of information flow propagation. We use `Smack` [Rakamarić and Emmi 2014], a LLVM IR-based software model checker for the model checking tasks. `Smack` (develop branch; git commit 71e0ad02) performs bounded model checking and generates counterexamples. The loop unroll number is five.

Alias Analysis. Data and code pointers are used in the lifted LLVM IR code. In particular, LLVM IR jump tables are frequently leveraged since EVM supports indirect jump with opcode `JUMPI`. During the intra-procedural CFG recovery stage, we conservatively put all the legit successors of a jump table into a function, whenever JEB3 alerts us this function may use indirect jumps (in such cases `jump` statements can be found in the decompiled functions of JEB3). We note that the current implementation, while being sound, could lead to false positives (i.e., merging irrelevant jump table successors into one function; see our evaluation and discussion related to Fig. 12).

Some indirect memory accesses in LLVM IR (e.g., representing the `sload` and `sstore` EVM opcodes) would take a register to access memory. During the taint analysis stage, we do not perform any expensive point-to analysis. As for the model checking stage, the model checker itself (`Smack`) has a basic support for static pointer reasoning. Overall, we leave it as one future work to perform static analysis and precisely infer the possible value set of data and code pointers.

Speedup Model Checking. We use bounded model checking, where the `schedule` (Sec. 7.1) complexity is proportional to the number of public functions in the contract. Our observation is that the model checking takes more time when the contract program becomes more complex, which is intuitive. In Sec. 10, we explore the effects of code complex (code size, number of functions, etc.) on the performance of NPChecker.

Model checking leads to noticeable performance overhead for our analysis. Indeed, we spent considerable effort selecting and tuning the model checker; from all the model checkers we tentatively experimented, Smack [Rakamarić and Emmi 2014] fits our need best given its integration into the LLVM ecosystem and relatively mature support. The Smack front-end takes LLVM IR as the input and links with two well-developed engines, Boogie [Lahiri et al. 2009] and Corral [Haran et al. 2015], for verification tasks. Our empirical evidence shows that the Boogie backend is usually much faster than Corral, but Boogie does not have good support for generating counterexamples since it does not leverage debug annotations in LLVM IR.

And for Smack with the Corral [Haran et al. 2015] backend, we experimented with different configurations to speed up model checking, including the default counter example-guided abstraction refinement (CEGAR) which starts from a coarse abstraction by tracking a small subset of program variables and gradually adds more variables when encountering infeasible counterexamples, and also the "trackAllVars" option which keeps track of all the variables in the first place. Our empirical studies show that the later configuration is much faster in analyzing smart contracts, although sometimes it is slower than CEGAR. Without further insight on how to fine tune the model checker engine and make it "adaptive", NPChecker is implemented to use Corral with the "traceAllVars" configuration for all the cases. Our experimental studies (and also suggested by the Smack developers) show that this configuration usually can lead to more efficient modeling.

Implementing the `get_auxiliary_var` Function. We now discuss the implementation of the `get_auxiliary_var` function used in the read/write checkers (Sec. 7.3.1). As mentioned before, EVM provides a persistent region to keep track of the global storage. In particular, fixed-size variables are laid out contiguously in the storage starting at zero while dynamically-sized data structures such as maps and arrays use `sha3` to find the starting position. As shown in the following case, the index of the statically-allocated variable `x` is fetched directly while the index of elements in `array` is computed with `sha3`. To compute the memory address for nested data structures, `sha3` function can be used for multiple times.

```
contract C {
  uint256 x; // memory address is 0
  uint256 [][] array;
  array[0].length; // memory address is sha3(1)
  array[1][1]; // memory address is sha3(sha3(1)+1)+1
}
```

Storage Indices. sha3 is the keccak256 hash function.

We create a ghost variable $\texttt{access}_i$ for each global variable. When a global variable is accessed via `sload` or `sstore`, we fetch its corresponding ghost variable and check for read-write hazards (see Sec. 7.3). We do this by creating a separate array to maintain all the ghost variables and searching for a corresponding ghost variable by mapping its storage index (the index can be acquired from the operand of `sload` and `sstore`) into our array index. The mapping algorithm is given below.

In general, we map all the variable indices (of fixed-size ones and `sha3` computed ones) into an array of 256 elements. In particular, we preserve the indices of fixed-size arrays (index starting at zero), while for indices of dynamic arrays (i.e., the `sha3` computed output), we only take the highest 8 bits. Although this approach may potentially lead to false positives, the collision rate is assumed low, given the limited number of global variables in each contract (usually much smaller than 256) and the good pseudo randomness of the crypto hash function `sha3`. Some related research has proposed techniques to recover the index of global variables [Grech et al. 2018]. We leave it as one future work to integrate their technique into NPChecker with additional engineering efforts.

```
get_auxiliary_var(idx) {
  // sha3 output is a 32-byte value
  if (idx is 32 bytes) {
  // we get the highest 8 bits
    idx = idx >> 248;
    return aux_array[idx];
  }
  // index of fixed-sized variable
  else return aux_array[idx];
}
```

Implementing the `merge` Function. To detect nondeterministic payments derived from cross-contract issues, we analyze the corresponding destination of each `delegate` call and extract the library smart contracts. The user and the library contracts are put together (see Fig. 8 for the usage of `merge`) and coordinated by one `schedule` function. We use JEB3 to infer the library address of a contract delegation. We note that for the delegation cases encountered during the evaluation, almost all the instances hard-code the delegation addresses in the code: we found 12 contracts (out of 3,075 distinct contract programs evaluated in this research) takes user-provided or dynamic computed address to access a library. We leave it as one future work to investigate these spurious contracts. Nevertheless, in case NPChecker cannot find the hard-coded address, it will resort to treat `delegate` as a normal `call`.

Table 5. Evaluation of new vulnerabilities or variants of common vulnerabilities. ✓ means the vulnerability is successfully detected while ✗ indicates the opposite.

	NPCHECKER	Securify	Oyente	Mythril
Constantinople reentrancy [ChainSecurity 2019]	✓	✓	✗	✗
cross-function reentrancy [evm 2018b]	✓	✓	✗	✗
delegation reentrancy [Rodler et al. 2018]	✓	✗	✗	✗
created-based reentrancy [Rodler et al. 2018]	✓	✗	✗	✗
funds transfer depends on `BlockNum`	✓	✗	✗	✗
funds transfer depends on `TxOrigin`	✓	✗	✗	✓

10 EVALUATION

We now present the evaluation of our research. Our dataset contains EVM bytecode of 30,000 online contracts. To collect the dataset, we randomly collected 10,000 contracts created in 2016, 2017, and 2018, respectively. We further deduplicated this dataset (as per their `sha256sum`) and obtained a set of 3,075 contracts.

Table 4. Statistics of the dataset used in evaluation.

Total # of distinct contracts	3,075
Total lines of LLVM IR code	34,484,470
Total # of functions	32,606
Total # of `call` statements	39,260
Total # of accesses of system properties	11,671
Total # of `sstore` statements	91,255
Total # of `sload` statements	392,611

To collect online contracts from the Ethereum mainnet, we use Google BigQuery to fetch addresses of all the contracts created in the past three years. We then iterated each contract address and downloaded its EVM bytecode from Etherscan [eth 2018]. Fig. 4 presents some statistics of the dataset used in evaluation, in terms of the code size, number of functions (we only count the contract public functions recovered by JEB3). We also measured the number of external contract `call` statements, the accesses of system properties and global storage (via `sload` and `sstore` statements).

Comparison with Existing Work. Table 2 lists related work in this field, and we are able to access five of these tools, which are Oyente [Luu et al. 2016], Mythril [ConsenSys 2018], Securify [Tsankov et al. 2018], MadMax [Grech et al. 2018], and ContractFuzzer [Jiang et al. 2018]. We take the first three tools since they are also static bug detectors which aim to detect similar bugs like our work does. MadMax has a different focus comparing to NPCHECKER, and ContractFuzzer [Jiang et al. 2018] conducts dynamic fuzz testing. ContractFuzzer generates test inputs based on the ABI specifications of smart contracts. Therefore, it is limited to contracts with ABI present, which will drastically reduce the number of available contracts.

Oyente makes its exact version used in its paper available (via a docker container), and we use the latest version of Securify maintained on Github (git commit 7b2d3c5a on March 30, 2019). Mythril [ConsenSys 2018] is actively maintained by the Ethereum community, and we use its official docker image version (a snapshot released on March 20, 2019). Regarding our threat model, Oyente supports to check classic TOD and single-function reentrancy bugs (belonging to $\boldsymbol{NP}_{II}$). It also supports to check timestamp dependence (one kind of $\boldsymbol{NP}_{I}$ bug). Mythril checks reentrancy bugs ($\boldsymbol{NP}_{II}$ bug) but does not support TOD bugs. It can check $\boldsymbol{NP}_{I}$ bugs by capturing dependencies on multiple system properties. Securify has a comprehensive support for $\boldsymbol{NP}_{II}$ including several TOD and reentrancy variants. However, it does not support any $\boldsymbol{NP}_{I}$.

While none of the available tools can check "Failed Call" bugs (see Sec. 3.2 for a description of this pattern), we note that both Mythril and Securify feature a bug pattern named "Unchecked Call" to check whether the return value of an external call is used by a branch condition. In general, we consider that these two bug patterns are *correlated*, since violating either pattern entails the return value is not properly handled. To present a fair comparison at our best effort, Securify and Mythril are deemed to find one $\boldsymbol{NP}_{I}$ bug whenever they report a finding of "Unchecked Call."

10.1 Case Study

Before launching experiments towards the real-world dataset, we start by implementing sample contracts containing recently-disclosed vulnerabilities or variants of common issues. As shown in Table 5, we report that NPCHECKER successfully identifies all vulnerabilities.

```
 1  contract PaymentSharer {
 2    mapping(uint256 => uint256) deposits;
 3    mapping(uint256 => address payable) first;
 4    mapping(uint256 => address payable) second;
 5
 6    function updateSplit(uint256 id, uint256 split) public {
 7      splits[id] = split;
 8    }
 9    function splitFunds(uint256 id) public {
10      address payable a = first[id];
11      address payable b = second[id];
12      uint256 depo = deposits[id];
13      deposits[id] = 0;
14
15      a.transfer(depo * splits[id] / 100);
16      b.transfer(depo * (100 - splits[id]) / 100);
17    }
18  }
```

Fig. 11. A (simplified) benign contract which becomes vulnerable after the Ethereum Constantinople upgrade.

Constantinople-reentrancy is a recently disclosed attack enabled by the Ethereum Constantinople upgrade [ChainSecurity 2019]. Consider the contract in Fig. 11. After the Constantinople update, it is feasible to perform a low-cost `sstore` to update program global state (line 7) during reentrancy from the external function call (line 16) and accordingly cause inconsistent program states [ChainSecurity 2019]. In contrast, existing pattern matching based detector [Tsankov et al. 2018] finds reentrancy bugs by only checking `sstore` after external calls (which does not appear in Fig. 11), since `sstore` was believed infeasible in the reenterred function before the Constantinople upgrade. NPChecker successfully flags a sample contract risky towards Constantinople reentrancy without any knowledge regarding this attack vector. Securify (the same team who first reported Constantinople reentrancy) is able to pinpoint this issue since Securify is presumably updated with this new pattern. In contrast, Oyente and Mythril do not feature knowledge to capture this reentrancy variant.

We also evaluate three advanced reentrancy attack vectors by implementing sample contracts of such vulnerabilities. As introduced in Sec. 3.5, subtler cross-function or even cross-contract reentrancy attacks are possible, and are generally ignored by existing static bug detectors since it requires more patterns to model. Indeed, besides Securify, which captures cross-function vulnerabilities, none of the vulnerabilities were detected by these tools. In contrast, we report that NPChecker can successfully flag bugs within these sample contracts since we systematically explore the feasibility of reentrancy attacks that can use contract delegation call and creation call. We also use two simple contracts which compute the amount of funds transfer by using either block properties `BlockNum` or transaction state `TxOrigin`. Since the per-block mining time varies, and contract (e.g., refunding within certain time frame) may cause confusions by using block number to decide a time frame. Also, as mentioned in Sec. 2.3.1, transaction origin is manipulable by an attacker. NPChecker provides comprehensive modeling of all the system and transaction properties and their influence on funds transfer. In contrast, besides Mythril, which features a pattern to flag dependencies on `TxOrigin`, other tools failed in pinpointing these two cases as vulnerable.

10.2 Evaluation of Real-World Smart Contracts

Table 6 presents the overview of the evaluation results. In summary, we found a total of 1,111 buggy contracts, with only 113 (3.5%) analysis timeouts. We further categorize vulnerabilities found from this dataset into $\boldsymbol{NP}_I$ and $\boldsymbol{NP}_{II}$ bugs. NPChecker finds considerable amount of bugs belonging to both categories. We also report that NPChecker skips the analysis of 120 cases for which JEB3 throws decompilation exceptions.[5] Overall, we interpret the evaluation results as promising and

[5]We have reported our findings to the JEB3 developers and wait for their confirmation.

Table 6. Evaluation results overview. We report results on in total 3,075 unique contract programs out of 30K online contract instances. We have confirmed **all** the 120 analysis failure of NPChecker is due to decompilation error of JEB3 [PNF 2018]: no function information is recovered from the EVM bytecode. Note that a vulnerable contract reported in the second column could contain $\boldsymbol{NP_I}$ bug (the sixth column), $\boldsymbol{NP_{II}}$ bug (the seventh column), or both. The percentage is calculated by taking "3075" as the divisor.

Tool	# of Vulnerable Contract	# of Safe Contract	# of Analysis Timeout	# of Analysis Failure	# of Contracts with NP_I Bugs	# of Contracts with NP_{II} Bugs
NPChecker	1,111 (36.1%)	1,731 (56.4%)	113 (3.5%)	120 (3.9%)	738 (24.0%)	887 (28.8%)
Securify [Tsankov et al. 2018]	990 (32.2%)	1,667 (54.2%)	0	418 (13.6%)	418 (13.5%)	909 (29.6%)
Oyente [Luu et al. 2016]	353 (11.5%)	2,722 (88.5%)	0	0	39 (1.3%)	346 (11.3%)
Mythril [ConsenSys 2018]	1,111 (36.1%)	1,964 (63.9%)	0	0	654 (21.3%)	580 (18.9%)

reasonable; many online contracts contain payment bugs due to the developers' unawareness of the unique Ethereum execution model.

As aforementioned, we compare NPChecker with two state-of-the-art analyses and the de facto static contract analyzer developed by the Ethereum community. We report their evaluation results in Table 6 as well. JEB3 reports decompilation error for 418 cases, while Mythril do not throw any exception. We note that when analyzing certain contracts with Oyente, we observed a large number of "unknown instruction" messages. However, it seems that Oyente can still give analysis results even with such error messages (it never crashed). Hence we still consider Oyente has zero failure and report its evaluation results in this section. NPChecker outperforms all the other tools by finding more $\boldsymbol{NP_I}$ bugs. This is reasonable since Security cannot support to capture dependencies on Ethereum system properties (i.e., the block and transaction states), while Oyente only flags dependence on block property `Timestamp`. Mythril provides more bug patterns to help identify the `Timestamp` and `TxOrigin` dependence.

As for the comparison of $\boldsymbol{NP_{II}}$ bugs, we report to find comparable amount of $\boldsymbol{NP_{II}}$ bugs with Securify which features comprehensive sets of vulnerability patterns to check reentrancy and TOD related bugs, although some of their findings do not actually influence payments (hence would be deemed "safe" by NPChecker). Mythril cannot detect TOD related bugs and therefore potentially missed a number of issues. Oyente performs highly inconsistent with the other tools on our dataset since it is designed to pinpoint only single-function reentrancy and TOD issues of only one pair of two functions.

Processing Time. Our evaluation was conducted on a server machine with an Intel Xeon E5-2680 v4 CPU at 2.40GHz and 256GB of memory. The machine runs Ubuntu 18.04. We set the timeout for NPChecker as 60 minutes, and when executing other tools, we use their default settings and timeout threshold.

Table 7. Processing time.

Tool	Processing Time (CPU hours)
NPChecker	300.9
Securify	572.5
Oyente	111.5
Mythril	149.9
NPChecker with early termination condition on finding $\boldsymbol{NP_I}$	169.6

Table 7 reports the processing time of each tool. NPChecker takes over 300.9 CPU hours to process the whole dataset: on average NPChecker takes 5.8 CPU minutes to process one contract. We also report processing time of the other tools. Securify takes 572.5 CPU hours to finish the analysis. The other tools are relatively faster, given they are not comprehensive enough to capture relevant bug variants. Our observation is that the model checking tasks are computationally costly: the model checker becomes observably slower with the code size growing. Nevertheless, we consider it is reasonable to terminate the analysis and raise implications for manual investigation, whenever the more efficient analysis of $\boldsymbol{NP_I}$ bugs (recall only taint analysis is used to flag $\boldsymbol{NP_I}$ bugs) have reported positive findings. Therefore, we refine the Alg. 1 and add an early termination condition whenever a $\boldsymbol{NP_I}$ bug is found. We re-launched the whole experiments in terms of this new setting and we report that NPChecker takes 169.6 CPU hours to finish: actually the analysis of 738 contract programs can be finished earlier since they have $\boldsymbol{NP_I}$ bugs.

Our analysis adopts bounded model checking towards the lifted LLVM IR code, which could be slow when the analyzed programs become too complex. Fig. 12 reports breakdowns regarding

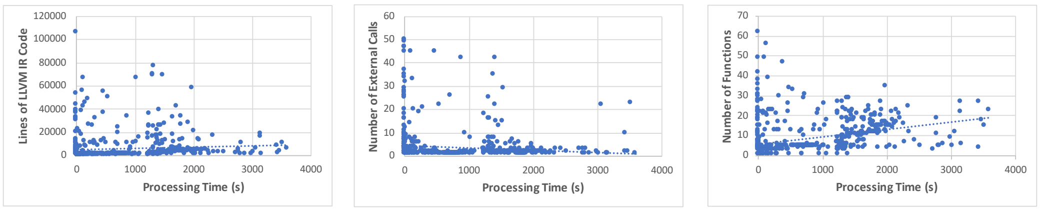

Fig. 12. Model checking performance breakdown.

how model checking performance changes w.r.t. the size of the code, the number of functions, and call operations. We interpret the results as intuitive: with the IR code becomes more complex, the model checking takes longer time. Particularly, the model checking time grows superlinearly to the number of public functions in the contract. As mentioned in Sec. 9, the complexity of the `schedule` function (Sec. 7.1) is proportional to the number of public functions in the contract.

At this step, we only report execution time where model checking returns *unsat* (see line 18 of Alg. 1). For such cases, model checking needs to comprehensively explore paths and therefore can paint a more accurate picture on the performance changes w.r.t. code complexity, comparing to "sat" cases where model checking can rapidly finish. Also, as discussed in Sec. 9, when recovering functions from the lifted LLVM IR, NPChecker does not perform expensive point-to analysis to resolve jump table destinations; instead, we conservatively put all valid successors of a jump table into a function whenever the decompiler implies indirect jumps are used in the function. As a result, irrelevant code blocks will be put into a function to inflate the generated IR code. Without knowing the ground truth, we leverage a simple strategy to remove presumably "inflated" IR code: given the observation that smart contracts are primarily succinct, we check the number of external calls in a contract's IR code and if there were over 50 external calls (highly unlikely; indicating an imprecise analysis of jump tables), we skip this contract. This step helps to eliminate 94 contracts from in total 1,305 contracts of "unsat" model checking results.

10.3 Manual Inspection

We perform further studies to confirm the findings of NPChecker. From our dataset, we randomly select 50 contracts with Solidity source code available on Etherscan (there are in total 409 contracts with source code available). We manually inspected the source code of each contract and confirmed whether they have vulnerabilities or not regarding our threat model. Similar to [Tsankov et al. 2018], we selected contracts with up to 300 lines of code to simplify the manual effort. Table 8 presents the comparison results. In general, we interpret the result as very promising; the proposed technique can faithfully flag buggy contracts with very low false positive and negative rates.

As aforementioned, Mythril and Securify use a different vulnerability pattern to analyze external call returns. NPChecker essentially tracks "Failed Call" pattern while both Mythril and Securify use a bug pattern named "Unchecked Calls" to detect whether the call return value is used as a branch condition. To present a fair comparison, we inspect each contract and use "Unchecked Call" to evaluate Mythril and Securify. Also, in Table 8, false positives (second row) are not necessarily equal to false positive $\boldsymbol{NP_I}$ (fourth row) plus false positive $\boldsymbol{NP_{II}}$ (sixth row), and similarly for false negatives. Consider a contract containing one $\boldsymbol{NP_I}$ bug but no $\boldsymbol{NP_{II}}$ bug, and a bug detector (e.g., Mythril) reports one $\boldsymbol{NP_I}$ bug from this contract, we will then consider it makes a true positive in flagging buggy contract (since it deems the contract as "vulnerable" and alerts users for investigation), but will have one false positive and one false negative for $\boldsymbol{NP_{II}}$ and $\boldsymbol{NP_I}$, respectively.

We observe that for both $\boldsymbol{NP_I}$ and $\boldsymbol{NP_{II}}$ bugs, Oyente missed many positive results. Indeed, Oyente only flagged one contract as vulnerable within our dataset. Mythril has a high false negative regarding $\boldsymbol{NP_{II}}$; this is reasonable since it does not analyze TOD bugs. Both Mythril and Securify have a number of false positives in $\boldsymbol{NP_I}$ bugs. Generally speaking, a contract is likely vulnerable, when it updates its storage after an external call, but is unaware whether the external call succeeds

Table 8. Manual inspection results. We report that out of 50 randomly selected and investigated contracts, we have 13 vulnerable cases and 37 safe cases. The full textual description of these inspected contracts can be found at https://www.dropbox.com/sh/90tm5drmeep9bqy/AAB0jKxkIevNct2eIvsYb7Oqa?dl=0.

	NPChecker	Oyente	Mythril	Securify
false positive	3	0	11	6
false negative	0	13	3	2
false positive ($\boldsymbol{NP_I}$)	2	0	7	5
false negative ($\boldsymbol{NP_I}$)	0	7	3	4
false positive ($\boldsymbol{NP_{II}}$)	2	0	6	5
false negative ($\boldsymbol{NP_{II}}$)	0	11	10	2

or not. Our manual study shows that the employed "Unchecked Call" pattern, checking whether the external call return is used by a path condition, leads to *false positives*. For instance, we find multiple cases where the external call is the last statement of a function. Although no path condition checks the call return (thus deemed as "vulnerable" regarding the "Unchecked Call" pattern), this actually does not lead to attack vectors.

We manually studied four false positives of NPChecker and we report that for those two false positives of $\boldsymbol{NP_{II}}$ bugs, they are due to the imprecise jump table analysis. Again, the lifted LLVM IR becomes "inflated" because the decompiler incorrectly guide us to merge a jump table which does not actually belong to this function. This increases the statements within the function (e.g., a function of three lines of Solidity code are translated into an IR function of over 1,500 statements). The inflated function generates incorrect read-write hazards on global variables.

Our manual investigation shows that there are two $\boldsymbol{NP_I}$ bugs (false positives), that were vulnerable, given our threat model, but are actually intended behavior of the contract. We show one example below (the other case has a similar pattern).

```
function sweepToken(address tokenContractAddress) returns (bool success) {
    ERC20 token = ERC20(tokenContractAddress);
    uint256 bal = token.balanceOf(this);
    require(bal > 0);
    return token.transfer(owner, bal);
}
```

where the "vulnerable" contract is a wallet which helps the contract owner to manage his ERC-20 tokens and `sweepToken` sweeps the full balance of a token to the owner's account. The first external call `balanceOf` gets the balance of addressed by `tokenContractAddress`. whose return value (i.e., balance) is checked by `require`. The second external call `transfer` will be executed to send `bal` amount of tokens to address `owner`, only when the token balance is not zero. While the given contract exposes "non-isolated" external `transfer` statements, and therefore entails a positive finding regarding our threat model (Sec. 3), this contract should be safe, by comprehending the intent of the given contract. Indeed, it is reasonable to stop token transfer and save transaction fee, in case the token balance is zero. Overall, our observation shows that for some cases, branch conditions or assertions which affect the execution of external calls are *intended*. We leave it as one future work to distinguish unintended non-isolation (presumably locks funds, as discussed in Sec. 3.2), or *intended* non-isolation. We present further discussion soon in Sec. 11.

11 DISCUSSION

"Failed Call" vs. "Unchecked Call". This research analyzes common vulnerabilities that can lead to nondeterministic payments in Ethereum smart contracts. In particular, the "Failed Call" pattern checked by the present and previous works [evm 2018b; Grech et al. 2018] asserts whether each payment is isolated and does not get affected by other payments (Sec. 3.2). In contrast, the "Unchecked Call" simply search for code fragments where the return value of external payments are checked by branch conditions which (arguably) indicates the return value is properly handled.

While the adopted threat model successfully helped to flag vulnerable contracts where the failure of one external payment "locks" the balance for further payments, still, we find cases where this pattern seems too aggressive and causes false alarms (see Sec. 10.3). Considering another example below, which forwards contract balance stored in mainDAO contract to the caller of `withdraw`:

```
contract WithdrawDAO {
  function withdraw() public {
    uint balance = mainDAO.balanceOf(msg.sender);
    if (!mainDAO.transferFrom(msg.sender, this, balance) ||
         !msg.sender.send(balance))
      throw;
  }
}
```

NPChecker reports to find $\boldsymbol{NP_I}$ bug, since the first payment call `transferFrom` determines the execution of `send` (i.e., violating "Failed Call"). However, as a payment forwarding contract, this pattern is actually intended, and the "Unchecked Call" pattern would entail this contract as "safe." Overall, both "Failed Call" and "Unchecked Call" seem to be vaguely defined, and to some extent, they assert *contradict* facts. As discussed in Sec. 3.6, the primary usage of NPChecker is to assist users by flagging potential defects for manual investigation. In addition, we leave it as one future work to explore the feasibility of rigorously defining the "Failed Call" pattern to promote bug detection or verification.

Agreement on Findings. Smart contract bug detectors could have a low agreement on their findings (due to the drifting of vulnerability patterns or threat models). A recent study [Perez and Livshits 2019] shows that when running Securify and Zeus [Kalra et al. 2018] regarding the same dataset to identify reentrancy bugs, their agreement is less than 1% (Securify and Oyente are reported to have 23.9% overlaps). To understand the analysis agreement, we also measured how many vulnerable contracts flagged by different tools can overlap and reported our results in Fig. 13. In summary, NPChecker and Securify agree on 36.7% of the findings, while NPChecker and Mythril agree on 26.9%. Security and Mythril have a relatively low agreement of 22.7%. We interpret NPChecker can outperform the other tools by achieving a higher agreement, although none of these tools can extensively subsume others' findings. We admit that it is challenging to propose a unified threat model, given the diverse and obscure intention of contracts in the wild. NPChecker is currently implemented to detect payment-related issues. We leave it as one future work to explore other threat models within the proposed framework.

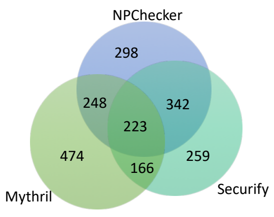

Fig. 13. Agreement among tools for flagging vulnerable contracts. We skip the analysis of Oyente since it is a bit more inconsistent with others.

12 CONCLUSION

The goal of this work is to detect payment bugs in smart contracts due to the inherent nondeterminism of Ethereum and could presumably be leveraged by attackers and cause financial losses. We pinpoint nondeterministic variables and employ information flow tracking to check their influence on funds transfer. To expose various global variables under read-write hazards, we propose systematic instrumentation techniques to practically model a large subset of contract runtime behaviors. Our evaluation of 3,075 distinct contract programs resulted in promising findings; we identified 1,111 contracts containing payment bugs with reasonable cost.

ACKNOWLEDGMENTS

We thank the anonymous SPLASH/OOPSLA reviewers for their valuable feedback. Our special thanks go to the JEB3, Smack, and EVMJIT developers who provided us with much help, insight and advice. Chengyu Zhang was partially supported by the China Scholarship Council, and NSFC Projects No. 61572197 and No. 61632005.

REFERENCES

2018. Etherscan.IO. https://etherscan.io.

2018a. EVMJIT. https://github.com/ethereum/evmjit.

2018b. Known Attacks of Ethereum Smart Contract. https://consensys.github.io/smart-contract-best-practices/known_attacks/.

2019. Provable: Provable^{TM} Random Number Generator. http://provable.xyz/.

2019. RANDAO: A DAO working as RNG of Ethereum. https://github.com/randao/randao/blob/master/README.md.

2019. Securify Git Issues. https://github.com/eth-sri/securify/issues/98.

Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. 2018. Towards verifying ethereum smart contract bytecode in Isabelle/HOL. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. ACM, 66–77.

Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, et al. 2016. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*. ACM, 91–96.

ChainSecurity. 2019. Constantinople enables new Reentrancy Attack. https://medium.com/chainsecurity/constantinople-enables-new-reentrancy-attack-ace4088297d9.

Code4Block. 2018. CVE List Found by Team Code4Block. https://github.com/TEAM-C4B/CVE-LIST.

ConsenSys. 2018. Mythril Classic. https://github.com/ConsenSys/mythril-classic.

Yu Feng, Emina Torlak, and Rastislav Bodik. 2019. Precise Attack Synthesis for Smart Contracts. arXiv:cs.CR/1902.06067

Geth. 2018. Go Ethereum. https://geth.ethereum.org/downloads/.

Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. MadMax: Surviving Out-of-gas Conditions in Ethereum Smart Contracts. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 116 (Oct. 2018), 27 pages.

Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. A semantic framework for the security analysis of ethereum smart contracts. In *International Conference on Principles of Security and Trust*. Springer, 243–269.

Arvind Haran, Montgomery Carter, Michael Emmi, Akash Lal, Shaz Qadeer, and Zvonimir Rakamarić. 2015. SMACK+Corral: A Modular Verifier (Competition Contribution). In *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (Lecture Notes in Computer Science)*, Christel Baier and Cesare Tinelli (Eds.), Vol. 9035. Springer, 450–453.

Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, et al. 2018. Kevm: A complete formal semantics of the ethereum virtual machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE, 204–217.

Yoichi Hirai. 2017. Defining the ethereum virtual machine for interactive theorem provers. In *International Conference on Financial Cryptography and Data Security*. Springer, 520–535.

Bo Jiang, Ye Liu, and W. K. Chan. 2018. ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*. 259–269.

Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts. In *Proceedings of the 2018 Network and Distributed Systems Security (NDSS) Symposium (NDSS '18)*.

Aashish Kolluri, Ivica Nikolic, Ilya Sergey, Aquinas Hobor, and Prateek Saxena. 2018. Exploiting The Laws of Order in Smart Contracts. arXiv:cs.CR/1810.11605

Johannes Krupp and Christian Rossow. 2018. teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, 1317–1333.

Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakamarić. 2009. Static and Precise Detection of Concurrency Errors in Systems Code Using SMT Solvers. In *Proceedings of the 21st International Conference on Computer Aided Verification (CAV '09)*. 509–524.

Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, 254–269.

Anastasia Mavridou and Aron Laszka. 2018. Tool demonstration: FSolidM for designing secure Ethereum smart contracts. In *International Conference on Principles of Security and Trust*. Springer, 270–277.

Satoshi Nakamoto et al. 2008. Bitcoin: A peer-to-peer electronic cash system. (2008).

Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, 653–663.

Daniel Perez and Benjamin Livshits. 2019. Smart Contract Vulnerabilities: Does Anyone Care? arXiv:cs.CR/1902.06710

PNF. 2018. JEB Decompiler. https://www.pnfsoftware.com/.

Shaz Qadeer and Dinghao Wu. 2004. KISS: Keep It Simple and Sequential. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI '04)*. ACM, 14–24.

Zvonimir Rakamarić and Michael Emmi. 2014. SMACK: Decoupling Source Language Details from Verifier Implementations. In *Computer Aided Verification*, Armin Biere and Roderick Bloem (Eds.). 106–113.

Michael Rodler, Wenting Li, Ghassan O. Karame, and Lucas Davi. 2018. Sereum: Protecting Existing Smart Contracts Against Re-Entrancy Attacks. *CoRR* abs/1812.05934 (2018).

SECBIT. 2018. Awesome Buggy ERC20 Tokens. https://github.com/sec-bit/awesome-buggy-erc20-tokens.

Ilya Sergey and Aquinas Hobor. 2017. A Concurrent Perspective on Smart Contracts. In *Proceedings of the 1st Workshop on Trusted Smart Contracts*.

David Siegel. 2016. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.

Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. 67–82.

D. Wood. 2014. Ethereum: a secure decentralised generalised transaction ledger.