EtherClue: Digital investigation of attacks on Ethereum smart contracts

Simon Joseph Aquilina[a], Fran Casino[c,d], Mark Vella[a,*], Joshua Ellul[a,b], Constantinos Patsakis[c,d]

[a]*Department of Computer Science, University of Malta, Msida, Malta*
[b]*Centre for Distributed Ledger Technologies, University of Malta, Msida, Malta*
[c]*Department of Informatics, University of Piraeus, 80 Karaoli & Dimitriou str., 18534 Piraeus, Greece*
[d]*Information Management Systems Institute, Athena Research Center, Artemidos 6, Marousi 15125, Greece*

Abstract

Programming errors in Ethereum smart contracts can result in catastrophic financial losses from stolen cryptocurrency. While vulnerability detectors can prevent vulnerable contracts from being deployed, this does not mean that such contracts will not be deployed. Once a vulnerable contract is instantiated on the blockchain and becomes the target of attacks, the identification of exploit transactions becomes indispensable in assessing whether it has been actually exploited and identifying which malicious or subverted accounts were involved.

In this work, we study the problem of post-factum investigation of Ethereum attacks using Indicators of Compromise (IoCs) specially crafted for use in the blockchain. IoC definitions need to capture the side-effects of successful exploitation in the context of the Ethereum blockchain. Therefore, we define a model for smart contract execution, comprising multiple abstraction levels that mirror the multiple views of code execution on a blockchain. Subsequently, we compare IoCs defined across the different levels in terms of their effectiveness and practicality through *EtherClue*, a prototype tool for investigating Ethereum security incidents. Our results illustrate that coarse-grained IoCs defined over blocks of transactions can detect exploit transactions with less computation; however, they are contract-specific and suffer from false negatives. On the other hand, fine-grained IoCs defined over virtual machine instructions can avoid these pitfalls at the expense of increased computation which are nevertheless applicable for practical use.

Keywords: Blockchain Forensics, Ethereum Attacks, Indicators of Compromise, Ethereum Operational Semantics

---

*Corresponding author
   Email addresses: `simon-joseph.aquilina.15@um.edu.mt` (Simon Joseph Aquilina), `francasino@unipi.gr` (Fran Casino), `mark.vella@um.edu.mt` (Mark Vella), `joshua.ellul@um.edu.mt` (Joshua Ellul), `kpatsak@unipi.gr` (Constantinos Patsakis)

1. Introduction

Nowadays, Ethereum is the largest public blockchain supporting smart contract execution. While introducing flexibility in terms of programmable transactions, as defined by Turing complete programming languages, the consequence is an enlarged attack surface [1, 2]. The immediate venue for attacks is presented by the smart contract code itself whenever it deviates from intended functionality. This problem is aggravated by incomplete knowledge of how a smart contract is executed at runtime by inexperienced programmers. Security vulnerabilities at this level can be similar to those found in non-blockchain code, e.g. arithmetic overflows, reentrancy, and poor randomness. Others are specific to the particular nature of smart contracts, e.g. call to unknown third-party contract code, improperly checking the return status of external calls, and denial of service (DoS) vulnerabilities related to gas-based mitigation of infinite execution bugs or call-stack depth limits, among many others. Furthermore, the number of currency/token-based contracts deployed on Ethereum aggravate the classic vulnerability of insufficiently sanitised user-controlled inputs.

Other entry points for attacks are presented by the runtime environment supporting contract code execution. Ethereum's infrastructure requires a virtual machine for code execution, while the consensus protocol's complexity is increased as a result of taking into account contract execution during block mining. The case of malicious miners being selective of which transactions, or in which order, they are included inside a mined block, is one such example. By doing so, an attacker can exploit time or transaction order-dependent logic inside smart contracts. Finally, Ethereum's external environment comprises code that interacts with Ethereum nodes, e.g. dApps and off-chain crypto wallets, further extend the attack surface [3]. At this level, attacks exploit vectors within the technologies in question, e.g. DOM-based XSS for dApps, or spyware in the case of desktop/mobile applications. Ultimately, these attacks can tamper with the submitted transactions in various ways.

In recent years, we have witnessed a number of notorious security incidents concerning the Ethereum blockchain that resulted in significant financial losses or drastic measures being taken to avoid consequences. The DAO crowd-funding platform incident is perhaps one of the most representative of these, as on June 18th, 2016, attackers managed to transfer \$60M under their control [4]. The funds were salvaged only through a hard fork, a measure which goes directly against the principle of allowing autonomous authorities to be arranged foregoing central authority [5]. In April 2018, an integer overflow vulnerability was the root cause for the attack on BECToken, resulting in the theft of 10^{58} tokens. Just earlier this year, a potential multi-million dollar disaster for users of Defi Saver was only averted by the timely action of its development team when notified of a vulnerability in one of their digital asset management contracts [6].

This work aims to identify exploit transactions and, therefore, the focus is on vulnerabilities that can be exploited solely via maliciously crafted transactions. These are submitted either directly from an externally owned account, i.e. an account not associated with code, or indirectly through a malicious smart contract. Ideally, such vulnerabilities would have been detected prior to contract instance creation, even more so when considering blockchain immutability. Therefore, tools that perform static analysis of contract code, identifying security vulnerabilities inside the source code (e.g.

2

Solidity) or decompiled Ethereum Virtual Machine (EVM) bytecode of smart contracts, comprise the most common approaches. Notable examples include Oyente [7], analysis tools based on Gigahorse [8], as well as EtherTrust[9] and Securify [10]. Others take a testing approach and focus on generating test inputs using a fuzzing approach [11, 9, 12], comparing test outcomes with test oracles. However, in both approaches, comprehensive execution path coverage remains an open challenge. As a result, vulnerable contracts may still be deployed on the blockchain. In such cases, exploit transactions can be identified *post-factum* during a subsequent investigation with the help of Digital Forensics and Incident Response (DFIR) tools. Investigations of this nature can help, first and foremost, to understand whether the smart contract in question has been exploited. If in the affirmative, the identification of exploit transactions provide links to the source accounts. These accounts may either belong to the perpetrators or victims whose accounts have been compromised. Furthermore, the linked accounts may be associated with further smart contracts, which at this point would have to be considered malicious, with any future transactions originating from them considered suspicious.

Indicators of Compromise (IoC) constitute evidence of intrusion and are typically directly related to the long-term side-effects of successfully executing malware [13]. In the case of WannaCry ransomware attacks on a Windows machine, for example, investigators can identify which artifact variant is in play, and therefore which recovery method could be effective, by checking whether the values `cmd.exe /c` and `C:\ProgramData\aucdehyopp032\tasksche.exe` can be found under the `"HKLM\System\CurrentControlSet\Services\ mssecsvc2.0"` key used for persistence. Another IoC can be based on the auto-propagation code that exploits EternalBlue. This step is observable within a network packet capture in the form of SMB traffic attempting a connection to `IPC$` and followed by an unusually large `NT Trans` SMB request. When one compares these host and network-level IoCs to Ethereum's state transitions, it is evident that we must consider radically different IoCs.

We approach the problem of responding to attacks on Ethereum smart contracts akin to responding to computer intrusions. In fact, during the earlier stages of the incident response process, a triage operation typically identifies compromised devices. This is a live forensics operation conducted while machines are still powered on. Indicators of Compromise (IoC) constitute forensic evidence of the intrusion and are the key elements of triage representing long-term residue of attack steps [13]. While similar IoCs could have been considered for attacks targeting Ethereum's operational environment, this approach does not apply at the smart contract level. If we consider contract execution steps as a state machine, we would rather be dealing with state transitions that update Ethereum blocks, add transactions to mined blocks and manipulate the EVM's internal state. Therefore, our first undertaking is precisely abstracting smart contracts execution in a model that considers all three levels. The result is the *EtherClue* DFIR tool (Figure 1), which given the address of a contract account and a vulnerable code location, it makes use of an IoC repository to identify potential exploit transactions from within an archive node which stores the full blockchain history.

Main contributions: The contributions of this work are manifold. More precisely, first, we introduce a model for Ethereum smart contract execution covering the block, transaction and EVM levels, and upon which IoCs for the integer under/overflow, DoS,
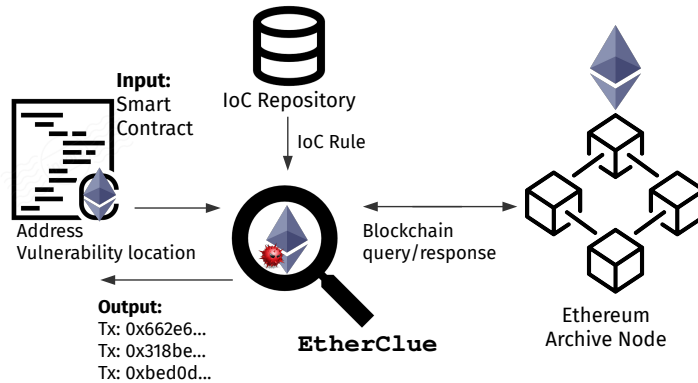
Figure 1: EtherClue DFIR tool.

and reentrancy exploit classes are defined. Furthermore, we introduce EtherClue, a new open-source DFIR tool making use of these IoCs.

Our experiments with EtherClue in both synthesised and real-world use cases and transactions. The full dataset for the synthesized cases has been publicly released, along with attack transaction labels for the real-world ones. Experiment results show that block-level IoCs can produce results quicker than at the EVM level. However, block-level IoCs are contract-specific and less precise than those defined at the EVM level. Therefore, a balance of precision and speed depending on the application must be further assessed. Furthermore, we conducted some Ethereum mainnet case studies, demonstrating EtherClue's practicality for EVM-level IoCs.

This paper is structured as follows: Section 2 provides an overview of Ethereum from the perspective of smart contract execution, as triggered by transactions, as can be observed at the EVM/transaction/block levels; as well as positions EtherClue alongside existing tools useful for digital forensics and incident response concerning Ethereum and blockchain technology in general; Section 3 presents the proposed model of smart contract execution before proceeding with IoC definitions; Section 4 highlights the main components of EtherClue's prototype implementation, and which is subsequently used for experimentation in Section 5. Section 6 concludes this work by highlighting the implications of the results produced using our public dataset, as well as indicating the next steps required to bring EtherClue closer to a production-ready state.

2. Background

Defining IoCs for smart contract attacks requires a characterization of their blockchain execution context. When viewing Ethereum as a world computer [14], contracts become the programs that manipulate its global state and whose integrity is safeguarded by mainnet, Ethereum's live blockchain. Mainnet is kept synchronized and immutable by a combination of consensus and block hash chaining between participat-

4

ing nodes. They expose functions that, when called, as a result of transaction execution during block mining, manipulate this global state. Transactions, in turn, have their authenticity safeguarded by public-key cryptography. IoCs can either focus solely on this global state as a forensic source or else may also consider the intermediate states resulting from individual transactions' execution, down to the individual EVM instructions as candidate forensic sources.

2.1. Ethereum Transactions and Smart Contracts

The most basic view of Ethereum is that of a sequence of accounts holding a value in cryptocurrency, characterizing Ethereum's global state. Accounts are identified by a 160-bit address linked to a private key used to authenticate all currency transfers requested on their behalf. *Ether*, Ethereum's cryptocurrency, transfers between these so-called *external accounts* take the form of transactions. Transactions identify the source/destination addresses, the value in Ether, along with a digital signature authenticating the request, among other fields. Smart contracts provide a way to define custom ways of moving Ether between accounts, enabling the development of sub-currencies (token), wallets, autonomous governance, and decentralized gambling/lottery applications [15], among others.

Listing 1 shows a simplified wallet smart contract, Wallet.sol, written in Solidity [16]. A contract is itself an account on the blockchain, also holding a value in Ether, with the important difference that it includes code. Wallet.sol exposes two functions, deposit() (lines 5-12) and withdraw() (lines 9-18), both of which are callable by transactions with the contract's address as destination. The former is marked as payable, meaning it can receive an ether transfer and which payment (msg.value) is recorded in the money (permanent) storage variable (line 3). The withdrawal function accepts an amount argument as part of the transaction's call data, resulting in a further internal message call (line 12), or at least if sufficient funds are available for the transaction's source address (msg.sender).

Listing 1: Wallet.sol - A Solidity wallet smart contract.

```solidity
1 pragma solidity >=0.5.8;
2 contract Wallet {
3   mapping(address => uint) money;
4   /* Deposit ether with the contract-account. */
5   function deposit() public payable {
6     money[msg.sender] += msg.value;
7   }
8   /* Withdraw ether from the contract-account. */
9   function withdraw(uint amount) public {
10    uint b = money[msg.sender] - amount;
11    if (b > 0) {
12      (bool rv, ) = msg.sender.call{value: amount}("");
13      if (rv)
14      {
15          money[msg.sender] = b;
16      }
17    }
18  }
19 }
```

Listing 2: Bytecode snippet from the compiled `Wallet.sol`.

```
1 ...SNIP...
2 SUB   //money[msg.sender] - amount
3 SWAP1   //uint b = money[msg.sender] - a...
4 POP   //uint b = money[msg.sender] - a...
5 PUSH 0  //0
6 DUP2  //b
7 GT     //b > 0
8 ...SNIP...
9 GAS   //msg.sender.call{value: amount}...
10 CALL  //msg.sender.call{value: amount}...
11 ...SNIP...
```

Prior to blockchain deployment and execution of smart contracts have to be compiled to EVM bytecode and possibly assigned an initial value in Ether via one more transaction. The EVM is a stack machine, as indicated by the typical stack-manipulating opcodes on lines 2-6 of the compiled `Wallet.sol` shown in Listing 2. Lines 9-10 concern the internal message call issued to perform withdrawal via the `CALL` opcode. Of particular interest is the `GAS` opcode. This instruction reflects Ethereum's gas system, where instruction execution and memory utilization consume gas paid in Ether by its caller account.

2.2. Ethereum State Transitions

When considering Ethererum as a state machine, the execution of `Wallet.sol` triggers state transitions on three levels: EVM instruction, transaction, and block mining. At the EVM-level, Figure 2, the contract's persistent storage and EVM's volatile memory sections characterize the execution state, while EVM instructions trigger transitions. A new state is created at the start of all external and (for some) internal calls. A read-only data-section stores the call's arguments, other transaction data, e.g. sender's address, transaction value in Ether, as well as block-related context, e.g. block's hash and timestamp. The contract storage is also made available, in our example storing `money`'s key-value pairs. Any changes to it will eventually be reflected in the global state post-successful execution. The stack area is the most dynamic, with Figure 3 illustrating a state transition triggered by a `SUB` instruction, acting on the top of the stack during `withdraw()`'s execution.

The transaction level's state, Figure 3, is at a higher level of abstraction and is oblivious of the EVM level. Rather, it is concerned with transactions updating Ethereum's global state. This state is an array of externally owned and contract accounts, with the only difference with the latter being associated with contract code and their persistent storage variables. Transaction-level state transitions, comprising the initial transaction and any internal ones triggered as a consequence, manipulate accounts through transaction fields (e.g. destination account address and the cryptocurrency value fields) and contract code execution. In the case of a transaction calling `Wallet.sol`'s withdrawal function for 1 Wei (1 Ether = 10^{18} Wei), the post-state reflects an updated value in the `money` map.

Transactions are executed in blocks, and they are stored as an immutable blockchain to safeguard the global state's integrity. At the block level, Figure 4, execution states are defined by these newly accepted blocks of transactions (Tx) and

EVM state *i*

Code		Stack (volatile)	
169	SUB	0:	0x02
170	SWAP1	1:	0x01
171	POP	2:	0x00
172	PUSH 0	3:	0x01

Call Data (volatile – read only)	
0:	0x2e1a7...00001

Contract storage (persistent)	
0xac40..	Key: 0x1a10 ...f02 Value: 0x02

EVM state *i+1*

Code		Stack (volatile)	
169	SUB	0:	0x01
170	SWAP1	1:	0x00
171	POP	2:	0x01
172	PUSH 0	3:	0x5a

Call Data (volatile – read only)	
0:	0x2e1a7...00001

Contract storage (persistent)	
0xac40..	Key: 0x1a10 ...f02 Value: 0x02

SUB

Key: 0x1a10 ...f02
Value: 0x02

Key: 0x1a10 ...f02
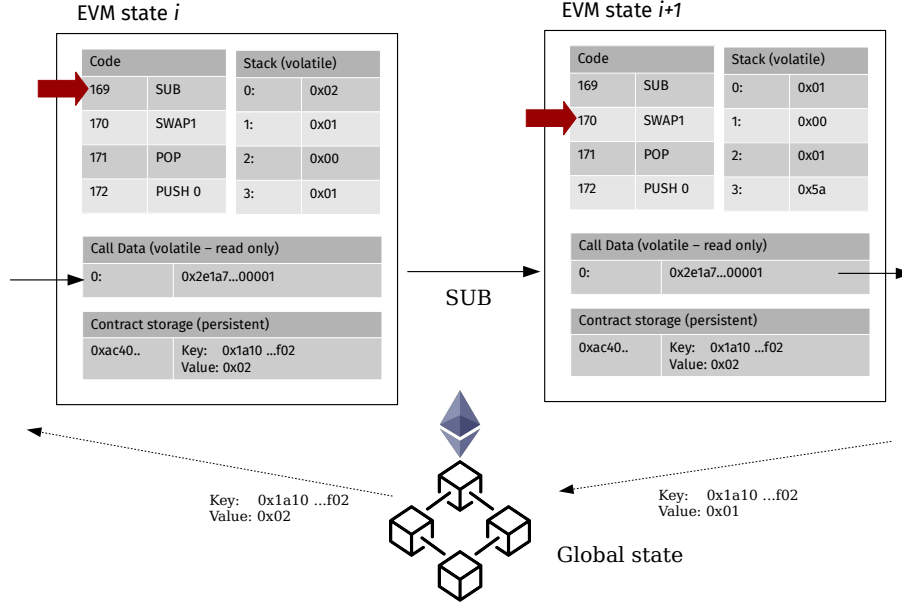Value: 0x01

Global state

Figure 2: An EVM-level state transition triggered by instruction execution.

associated block meta-data, e.g. the block hash and the hash of the previous block chained to it. This level does not keep account of intermediate states generated by individual transactions or EVM-level state transitions, except for values inside transaction receipts (Rx), or log entries made by contract code. Nevertheless, through transaction tracing, typically offered by Ethereum nodes [17], it is possible to replay transaction execution up to EVM instruction level. This is achieved by recomputing (or caching) the global state at the transaction call point through blockchain traversal and accessing the contract bytecode stored as part of the global state. The global state's integrity is safeguarded by a hash, called the state root, stored as part of every block.

2.3. Notation and Execution Semantics

The semantics underpinning Oyente's symbolic execution engine can provide the required model of smart contract execution [7]. They build upon the formalism presented in the Ethereum Yellow Paper [14]; referred to from now on as EYP, where the *global state* made up of a sequence of all external/contract accounts is denoted as σ. In contrast, the state of a single account with address γ is denoted as $\sigma[\gamma]$. A single external/contract account state $\sigma[\gamma]$, is the tuple $\langle balance, nonce, storageHash, codeHash \rangle$. The first two tuple elements correspond to the current value in currency owned by the account and countermeasure for replay attacks on digitally signed transactions. The last two are only relevant for contract accounts. They correspond to the root hashes of a (modified) Merkle trie structure (a key/value store with integrity protection) [14]
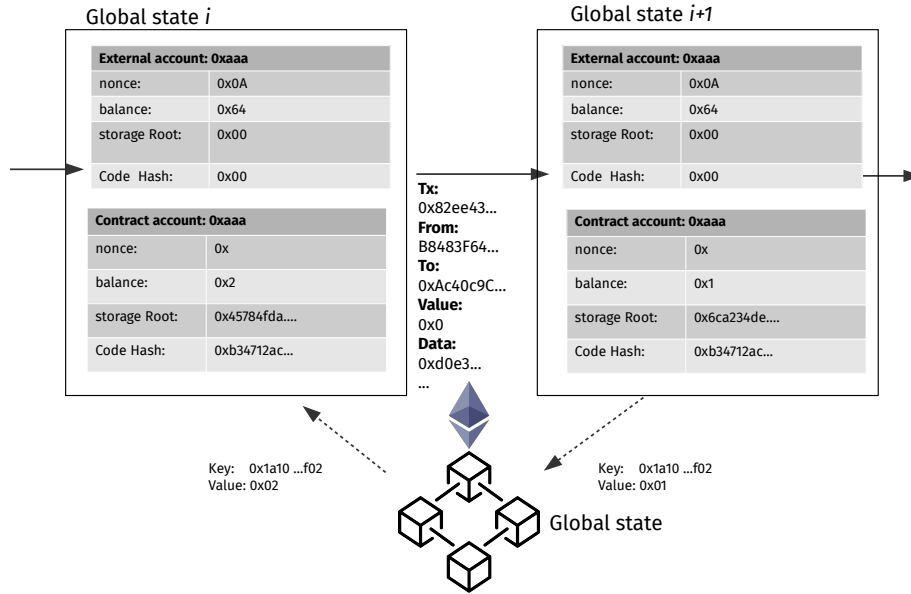
7

Global state *i*

External account: 0xaaa	
nonce:	0x0A
balance:	0x64
storage Root:	0x00
Code Hash:	0x00

Contract account: 0xaaa	
nonce:	0x
balance:	0x2
storage Root:	0x45784fda....
Code Hash:	0xb34712ac...

Tx:
0x82ee43...
From:
B8483F64...
To:
0xAc40c9C...
Value:
0x0
Data:
0xd0e3...
...

Global state *i+1*

External account: 0xaaa	
nonce:	0x0A
balance:	0x64
storage Root:	0x00
Code Hash:	0x00

Contract account: 0xaaa	
nonce:	0x
balance:	0x1
storage Root:	0x6ca234de....
Code Hash:	0xb34712ac...

Key: 0x1a10 ...f02
Value: 0x02

Key: 0x1a10 ...f02
Value: 0x01

Global state

Figure 3: A transaction-level state transition triggered by transaction execution.

Newly mined block

Block *n-1*
(Global state *i*)

Block *n*
(Global state *i+t*)

Block: 0xba4f62...		
Prev. Hash	TxRoot	Receipts Root
...		State Root

Block: 0x45def82...		
Prev. Hash	TxRoot	Receipts Root
State Root		...

Guarantees
State
Integrity

Tx: i-4
Tx: i-3
Tx: i-2
Tx: i-1
Tx: i

Tx: i+1
Tx: i+2
Tx:82ee43...
Tx: i+3
Tx: i+t

Global state
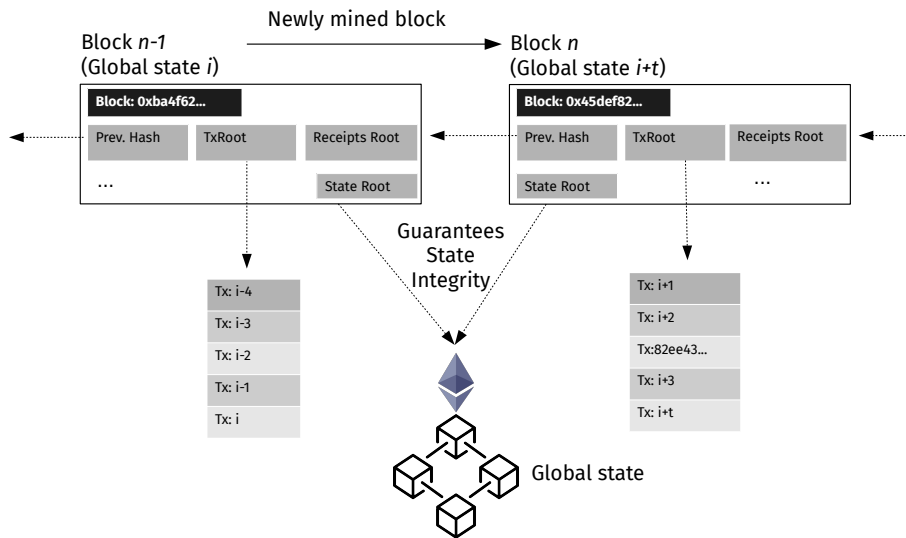
Figure 4: A block-level state transition triggered by block mining.

storing all contract persistent variables and the hash of the smart contract code, respectively.

Accounts states are accessible via the *world state* lookup function, $\gamma \rightarrow \sigma[\gamma]$, in turn requiring access to the blockchain *BC* to verify account state integrity. The three levels of execution, at the EVM/transaction/block levels are defined by the semantics of small-step evaluation ($\rightsquigarrow$), successful transaction execution ($\xrightarrow{T}$) and big-step evaluation ($\Downarrow$), updating the EVM state (μ), σ and *BC* respectively. Operational semantics rules are used, each describing execution at the EVM, transaction or block levels of abstraction, and structured as follows:

$$\text{Execution:} \frac{\begin{array}{c} expression_1(\mu|\sigma|BC, \mu'|\sigma'|BC') \\ expression_2(\mu|\sigma|BC, \mu'|\sigma'|BC') \\ ... \\ expression_n(\mu|\sigma|BC, \mu'|\sigma'|BC') \end{array}}{\mu|\sigma|BC \quad \rightsquigarrow | \xrightarrow{T} | \Downarrow \quad \mu'|\sigma'|BC'}$$

where expressions describe how post-states are evaluated from pre-sates and define any constraints on the transition. At the EVM level, the semantics of a subset of the EVM, called `EtherLite`, is provided. Combined, these semantics inform the implementation of a symbolic execution engine. In turn, Oyente benefits from a symbolic execution-based approach to inspecting contracts on a path-by-path basis. For each path, a number of instruction sequence-centric conditions are applied to detect vulnerabilities. This approach is superior compared to dynamic testing, eliminating the need for input generation to maximise code coverage. However, accurate emulation of runtime execution becomes critical, warranting the need for precisely defined execution semantics.

2.4. Related work

Post-factum detection and recovery from smart contract attacks is an area that has received much less attention as compared to proactive defences. While this is understandable, this problem cannot be ignored, especially when one considers the limitations of vulnerability detectors and the need to identify malicious or subverted accounts. DappGuard [18] is the earliest effort we found in this direction, with a focus on studying whether transactions and transaction receipt information extracted by blockchain exploration can help to uncover attacks. Its IoCs are referred to as attack fingerprints and compare to EtherClue's block-level IoCs. Similar approaches attempt to identify the discriminating features of accounts and associated transactions using machine learning [19, 20].

TxSpector [21], on the other hand, is the most recent related work and, in fact, carried out in parallel to EtherClue's study. It adopts a static analysis method to work on dynamic EVM bytecode traces instead and therefore compares to our EVM-level IoCs. Instruction traces enriched with data and control flow dependencies are derived using a modified Ethereum node. Manually written logic rules are used to identify those traces, at this point represented as logical relations, generated from known attacks. TxSpector can be seen to be a generalization of Sereum [22], which take a dynamic taint analysis approach to detect reentrancy exploit transactions. While Sereum was originally positioned as a real-time detector, implemented by customizing an Ethereum node, its underpinning technique could still be employed in a post-factum manner to

flag transactions in a similar fashion to TxSpector. Yet its scope is restricted to just reentrancy attacks and does not offer support for writing additional detector rules or IoCs, as per TxSpector or EtherClue, respectively.

Ultimately the goals of EtherClue and TxSpector are similar, however with one crucial difference. TxSpector, while capable of detecting exploit transactions, is actually conceived as a dynamic vulnerability detector, operating on instruction traces of already-executed transactions, as opposed to considering all possible execution paths that would have to be considered by a static analysis tool. On the other hand, EtherClue is positioned to complement vulnerability detectors, following up vulnerabilities detected within deployed contracts with an investigation concerning finding out whether the contract has yet been attacked along which accounts are implicated. The result is that of much simpler detection rules focusing on execution side-effects rather than program analysis-based ones. Furthermore, EtherClue's operational semantics foundation, which already underpins Oyente's [7] symbolic execution engine, can even provide a basis for automated IoC generation.

Other works have the potential to complement EtherClue during incident response. Firstly, the identified exploit transactions can be used to detect other attack transactions related to the same incident, while not necessarily being exploit transactions per se, e.g. transactions related to the setting up of malicious smart contracts or ones that transfer stolen currency to financial service contracts. DEFIER [23] is the only such tool we are aware of, whereby given a seed set of attack transactions, it performs data analysis using code/graph similarity, as well as time series analysis based on manually inspected attack kill-chains, to identify further incident-related transactions. Two other studies [24, 25] address the problem of the ever-increasing size of archive nodes and the implication to blockchain analysis. Some of the explored aspects include the application of parallel map/reduce and the practice of incomplete state storage that does not compromise high-fidelity transaction replay.

Once post-factum attack transactions are identified, the subsequent step is graceful recovery. One approach is to prepare for recovery during the smart contract programming stage using security patterns. The *Emergency Stop* pattern [26] involves deactivating critical contract functions when certain conditions are met, leaving only the possibility to withdraw funds. The *Virtual Upgrade* pattern [27] is less drastic, where critical functions are only exposed through a proxy contract that can dynamically route transactions to back-end contracts. This pattern provides flexibility, re-routing transactions to patched contracts at runtime. Automated patching can speed up this time-critical operation, reducing the opportunity window for attacks further still [28].

Other studies have a completely different scope than EtherClue's. Using honeypots for threat analysis, for example, can shed light on little-known attack techniques [29]. ContractLarva is a Runtime Verification tool that can prevent exploits from succeeding by instrumenting smart contracts with security property verification [30]. While this approach raises smart contracts' security level, it can only do so with respect to the defined properties and comes at the cost of additional gas consumption. Differently from incident response, the investigation of cryptocurrency-driven blockchains also concerns transactions related to fraud, extortion, money laundering and tax evasion during forensic accounting exercises [31, 32]. Finally, permissioned/private blockchains can be useful for chain-of-custody management of digital evidence [33, 34].

3. EtherClue IoCs

The model of smart contract execution encompasses all three levels of abstraction for state transitions, i.e. the EVM, transaction and block levels. The base state transitions build upon the operational semantics used for Oyente [7], while EtherClue's IoCs are defined in terms of additional constraints over the base definitions.

3.1. Model of smart contract execution

Oyente's [7] original semantics are primarily intended for accurate emulation, providing a suitable starting point for our model. Some modifications/additions are necessary to fit EtherClue's digital forensics needs and are as follows:

- Introduce a new *world state* map that supports the retrieval of past global states.

- Provide custom definitions for blocks, transactions, and related structures, based on those found in the EYP so as to focus solely on the forensic sources of interest.

- Likewise, abstract away details of the block mining process.

- Enhance the model with transaction receipts since these constitute evidence concerning individual transaction execution.

- Enhance EVM-level operational semantics with a provision to indicate the location of vulnerable code.

- EVM-level semantics are defined over grouped instruction classes, rather than for `EtherLite` (the sub-set of EVM instructions considered by Oyente). In this manner, a single IoC can be conveniently defined over multiple EVM instructions.

- Provide IoC templates for the EVM and block levels to detect exploit transaction during transaction replay.

We now enlist all definitions, the execution model underpinning EtherClue's IoCs, highlighting the parts specific to EtherClue explicitly by enclosing them into a rounded-boxed border.

The *global state* σ is an array of accounts, identifiable by the 160-bit contract address γ. We also adopt the $\sigma[\gamma]_s$ and $\sigma[\gamma]_c$ notation, to conveniently refer to the actual storage and code content respectively, rather than just the root hashes, and through which to access their internal elements, e.g. $\sigma[\gamma]_s[varname]$.

According to the EYP:

Definition 3.1. An Ethereum account $\sigma[\gamma]$ is defined as the tuple:

$$\sigma[\gamma] \stackrel{def}{=} \langle nonce, balance, storageHash, codeHash \rangle,$$

where *nonce* denotes a fresh number to use as countermeasure against transaction replay attacks; *balance* denotes the account balance in Wei $(= 10^{-18}$ Ether); *storagehash* denotes the root hash for the contract storage's Merkle trie, and *codeHash* denotes the hash for associated smart contract EVM code.

EVM level. Definitions at this level are expressed in the spirit of all smarts contracts executing in the context of the current global state σ. An *Activation Record Stack A* of *Activation Records AR* is used to implement the mechanism of internal transaction execution. *AR*s keep track of the EVM state μ to be resumed once the current nested transaction returns control to its caller. The *A* for an external transaction always starts empty (ε). Whenever an exception *EXC* is thrown, it is placed on the top of the stack, replacing the topmost *AR*, effectively returning control to its caller, that either handles *EXC* or bubbles it up further in the call stack. An *AR* reflects the μ that is to be reinstated following caller resumption, and therefore has to keep track of the executing code's address (only addition to Oyente's definition), the code itself and the next instruction to execute on resumption, along with stack and non-stack (e.g. call data) content. Therefore μ comprises the current *A* and the current σ in which it executes. State transitions on this level are described by small-step evaluations ($\rightsquigarrow$), and reflect the μs obtained for traced transactions. The full definition list for the EVM-level is the following:

Definition 3.2. We define an Activation Record (AR) as the tuple

$$AR \stackrel{def}{=} \langle id, M, pc, l, s \rangle,$$

where *id* denotes the γ of executing code, *M* denotes the smart contract's EVM code, *pc* denotes the program counter representing an offset into *M*, and *l, s* denote the non-stack memory data and stack memory data accordingly.

Definition 3.3. Activation Records *AR* are stored in Activation Records Stacks *A* defined as:

$$A \stackrel{def}{=} A^{normal} | EXC \cdot A^{normal}$$

$$A^{normal} \stackrel{def}{=} AR \cdot A^{normal} | \varepsilon$$

where:

- '$\cdot$': separates the top element; an *AR* or a thrown exception *EXC*, from the rest of the stack.

- A^{normal} denotes an activation record for 'normally halting' calls, while ε denotes the empty stack.

Definition 3.4. For an Activation Records stack *A* and an Ethereum account σ providing the execution context (see Definition 3.1), we define an EVM state μ as the tuple:

$$\mu \stackrel{def}{=} \langle A, \sigma \rangle$$

Definition 3.5. We abstract the transition from an EVM state μ to an EVM state μ' through the EVM opcodes, see for instance semantics of Table 1, and name it a *Small-step Evaluation* representing it as $\mu \rightsquigarrow \mu'$.

$M[pc]$	Conditions	μ	μ'
PUSH8 1	-	$\langle\langle id, M, pc, l, s\rangle, \cdot A, \sigma\rangle$	$\langle\langle id, M, pc+1, l, 1\cdot s\rangle \cdot A, \sigma\rangle$
PUSH8 5	-	$\langle\langle id, M, pc, l, 1\cdot s\rangle \cdot A, \sigma\rangle$	$\langle\langle id, M, pc+1, l, 5\cdot 1\cdot s\rangle \cdot A, \sigma\rangle$
SSTORE	$k = 5 \leftarrow pop(s)$ $v = 1 \leftarrow pop(s)$ $\sigma' \leftarrow \sigma[id][storage][k \rightarrow v]$	$\langle\langle id, M, pc+1, l, 5\cdot 1\cdot s\rangle \cdot A, \sigma\rangle$	$\langle\langle id, M, pc, l, s\rangle \cdot A, \sigma'\rangle$

Table 1: EVM-level transition semantics described over the EVM state μ.

Note that in Table 1, $a[i \mapsto v]$ evaluates to a new array identical to a, but storing the new value v at position i.

In EtherClue, we opted to define IoCs over grouped instruction classes and require the vulnerability's location(s) and associated parameters. This is both reasonable and practical since EtherClue's scope complements and does *not* replace vulnerability detectors. Furthermore, IoC definitions can focus solely on the identification of exploit side-effects, without being concerned with defining the nature of the vulnerability per se. The notation we make use of, $\otimes^{NAME}_{PARAM_{1...N}}$, identifies EVM instruction classes by *NAME*. The parameter list $PARAM_{1...N}$ provides a vulnerability-specific context for the instruction class in question. For example, $\otimes^{INT}$ groups multiple integer arithmetic instructions. In terms of Solidity source language constructs, this grouping ranges over all possible integer sub-types (`UINT8-256`, `INT8-256`). Therefore a parameter list is required to define the lower/upper bounds for specific sub-types. These parameters assume the availability of the (possibly decompiled) source code. On the other hand, $VULN\_LOCS$ is instantiated with a list of code offsets i, associated with the vulnerability in question, and where every opcode in $M[i]$ belongs to $\otimes^{NAME}$.

To provide a common template, EVM-level IoCs are defined as a sequence of constraint expressions, defined over μ, μ', and $VULN\_LOCS, PARAM_{1...N}$, qualifying those side-effects that render a small-step evaluation to be considered an exploit step. When all predicate expressions evaluate to true for a particular small-step evaluation, the corresponding transaction is flagged as an exploit during replayed execution. More formally:

Definition 3.6. We define an *EVM-level IoC template* as an EVM instruction class:

$$VULN\_LOCS \leftarrow M \text{ offsets}$$

$$\otimes^{NAME}_{PARAM_{1...N}} : \cfrac{\begin{array}{c} expression_1(\mu,\mu',VULN\_LOCS,PARAM_{1...N}) \\ expression_2(\mu,\mu',VULN\_LOCS,PARAM_{1...N}) \\ ... \\ expression_n(\mu,\mu',VULN\_LOCS,PARAM_{1...N}) \end{array}}{\mu \rightsquigarrow \mu'}$$

where:

- M offsets: a set of code offsets belonging to a detect vulnerability

- $VULN\_LOCS$: symbolic constant representing the above set of code offsets,

- μ:EVM-level pre-state

- μ':EVM-level post-state

- $\otimes^{NAME}$: an EVM instruction class that matches the IoC

- $PARAM_{1...N}$: EVM instruction class parameters

- *expression_i*: operational semantics expressions defined over $\mu,\mu',VULN\_LOCS,PARAM_{1...N}$, with an IoC match occurring whenever all predicate expressions evaluate to true

Transaction level. At this level, we consider the *sender*, *to*, *value* and *data* elements of successful transactions T as evidence sources. These elements correspond to the source/destination addresses, amount of cryptocurrency being transferred, and a data field identifying the contract's public function and arguments. Transactions are executed as a sequence of 0 (when a destination is also an external account) or more small-step evaluations $\rightsquigarrow^*$. State transitions $\xrightarrow{T}$ at this level operate on σ, transferring cryptocurrency corresponding to T's *value* to $\sigma[to]$'s *balance*. Further updates to σ may occur as a result of executing $\sigma[to]_c$ on termination of a small-step evaluation $\rightsquigarrow^*$. The finalized definitions comprise a sub-set of transaction attributes defined in the EYP and the definitions provided by Oyente for transaction execution, either ending successfully or with an exception. In the latter case, the currency associated with the transaction is not transferred, and the changes to the global state are not committed. Definitions 3.8 and 3.9, while maintaining the original semantics of transaction execution, are defined to reflect modification of their primitive definitions.

14

Definition 3.7. We define a Transaction T in terms of a subset of the EYP's definition, as the tuple $T \stackrel{def}{=} \langle sender, to, value, data \rangle$, where *sender* denotes sender's account address γ, *to* denotes the target's account address γ', *value* denotes the transaction amount in Wei, and *data* denotes the call data. That is an array comprising the hash of a public function exposed by the target *to* followed by the function arguments

Definition 3.8. Transaction Execution $\xrightarrow{T}$ resulting in successful completion:

$$\text{Tx-success:} \frac{\begin{array}{c} \langle sender, to, value, data \rangle \leftarrow T \\ M \leftarrow \sigma[to]_c \\ \sigma' \leftarrow \sigma[to][balance \mapsto (\sigma[to][balance] + value)] \\ \langle \langle to, M, 0, data, \varepsilon \rangle \cdot \varepsilon, \sigma' \rangle \rightsquigarrow^* \langle \varepsilon, \sigma'' \rangle \end{array}}{\sigma \xrightarrow{T} \sigma''}$$

where:

- σ represents the pre (global) state,

- σ'' represents the post (global) state,

- σ': represents the interim (global) state to store the new target account's balance prior to smart contract execution, and

- $\rightsquigarrow^*$ represents a sequence of 0 or more small-step evaluations ($\rightsquigarrow$), depending on whether the target address *to* is a smart contract account or not.

Definition 3.9. Transaction Execution $\xrightarrow{T}$ resulting in a raised exception:

$$\text{Tx-exception:} \frac{\begin{array}{c} \langle sender, to, value, data \rangle \leftarrow T \\ M \leftarrow \sigma[to]_c \\ \sigma' \leftarrow \sigma[to][balance \mapsto (\sigma[to][balance] + value)] \\ \langle \langle to, M, 0, data, \varepsilon \rangle \cdot \varepsilon, \sigma' \rangle \rightsquigarrow^* \langle EXC \cdot \varepsilon, \sigma'' \rangle \end{array}}{\sigma \xrightarrow{T} \sigma}$$

where:

- σ represents the pre (global) state,

- σ'' represents the post (global) state,

- σ' represents the interim (global) state to store the new target account's balance prior to smart contract execution, and

- $\rightsquigarrow^*$ represents the sequence of 0 or more small-step evaluations ($\rightsquigarrow$), depending on whether the target address *to* is a smart contract account or not.

While the transaction-level definitions are needed by the block-level ones, we deem IoCs at this level to be unnecessary. Detecting IoCs at this level would first and foremost accessing the state σ and σ'', upon which IoCs at this level would be defined. While transaction receipts at the block level (see Definition 3.12) can shed some light on the make-up of these states, their complete characterization is not. Rather, the only manner with which σ and σ'' can be computed is by executing $\leadsto^*$. Since this amounts to the same processing carried out at the EVM level, which also has access to σ and σ'', EVM-level detection trumps the transaction level at no additional cost.

Block level. This level is concerned with the execution of batched transaction sequences TXs, as chosen out of a transaction pool set Γ, during successful block mining. Successful transaction execution results in transaction receipts Rx (we only consider the log entries *logs* and transaction status codes *statusCode*) to be added to the block's receipt sequence RXs. A newly mined block β, therefore, performs a big-step evaluation, transitioning the global state σ_0 directly to σ_n after having batch-executed n transactions. Finally, β is appended to the blockchain BC, comprising a sequence of all the blocks mined so far. Overall we ignore details of the consensus protocol considered to not contribute as a forensic evidence source, e.g. details of the block mining process, such as proof-of-work, or the propagation of successfully mined blocks across the entire Ethereum peer network. However, we are not concerned with executing the next block of transactions here, but rather with being able to fetch any past σ_0/σ_n pair and verify whether the corresponding $\Downarrow$ constitutes a transition during which an exploit transaction occurred. To add this capability to our model, we enhance the notion of the World State ω mapping with the ability to take any state root (a hash of the global state corresponding to a block β) from the past and return its corresponding $\sigma_{stateRoot}$. This enhanced definition naturally also applies to EVM-level IoCs, and through which they obtain the ability to replay transaction traces in the first place. Therefore we say that EtherClue operates on an Archive Node $\langle BC, \omega \rangle$. The resulting definitions, omitting details unused by EtherClue and adding the concept of an archive node, are as follows.

First, we use Oyente's definition for a Transaction sequence:

Definition 3.10. A Transaction sequence TXs is an ordered set of n transactions

$$TXs \stackrel{def}{=} (T_1, \dots, T_n)$$

Definition 3.11. We define a Transaction Pool Γ as a sequence of transactions:

$$\Gamma \stackrel{def}{=} \{T_0, \dots, T_\infty\}$$

Definition 3.12. We define a Transaction Receipt R as a tuple:

$$R \stackrel{def}{=} \langle logs, statusCode \rangle,$$

where *logs* denote the event log entries as created by the EVM `LOGX` opcodes and *statusCode* denotes the transaction status code (success or failure).

Definition 3.13. We define a Receipt Sequence *RXs* as an ordered sequence of n Transaction Receipts:

$$RXs \stackrel{def}{=} (R_1, \ldots, R_n)$$

We use a subset of the Block definition from EYP as it is deemed relevant as a forensic source.

Definition 3.14. We define a Block β as the tuple:

$$\beta \stackrel{def}{=} \langle parent, TXs, RXs, stateRoot \rangle,$$

where *parent* refers to the block's parent block through its hash, *stateRoot* denotes the root hash of the Merkle trie storing σ, and *TXs*, *RXs* denote a Transaction and Receipt Sequence, respectively.

Definition 3.15. We define a Blockchain *BC* as the ordered finite set of n blocks:

$$BC \stackrel{def}{=} (\beta_0, \ldots, \beta_n)$$

Definition 3.16. We define a World State ω as:

$$\omega \stackrel{def}{=} stateRoot \mapsto \sigma_{stateRoot},$$

where $\sigma_{stateRoot} \colon \sigma$ for some block β with $\beta[stateRoot] = stateRoot$

Definition 3.17. We define an Archive Node $\mathcal{A}$ as a tuple of a blockchain *BC* and a world state ω:

$$\mathcal{A} = \langle BC, \omega \rangle$$

Definition 3.18. For a given Transaction Pool Γ from which a transaction sequence TXs is selected by a miner at its discretion, and using ε to denote and empty sequence and ... to delimit the first and last elements of a sequence; we define a *Big-step evaluation*, denoted as $\Downarrow$ in the context of blockchain BC, and generating the Receipt Sequence RX, as :

$$TXs \leftarrow select(\Gamma)$$
$$(T_1,...,T_n) \leftarrow TXs$$
$$Rx \leftarrow \varepsilon$$
$$\beta_{parent} \leftarrow last(BC)$$
$$\sigma_0 \leftarrow \omega[\beta_{parent}[stateRoot]]$$
$$\forall i, \quad 1 \le i \le n : \{$$
$$\sigma_{i-1} \xrightarrow{T_i} \sigma_i, Rx[i] \leftarrow generateReceipt(T_i)\}$$
$$\beta \leftarrow \langle \beta_{parent}, TXs, RXs, rootHash(\sigma_n)\rangle$$
$$\omega' \leftarrow \omega[\beta[stateRoot] \mapsto \sigma_n]$$

$$\text{BIG-STEP:} \frac{}{\langle BC, \sigma_0, \omega \rangle \Downarrow \langle BC \cdot \beta, \sigma_n, \omega' \rangle}$$

where σ_0, σ_n denote the global state prior/post β, respectively; $BC \cdot \beta$ denotes the appending of block β to BC; ω, ω' denote the world map updated to reflect β's state root; $select()$ denotes the miner's process of choosing transactions from Γ at its discretion; and $last()$ denotes the function that retrieves the last block (β) from blockchain BC; $generateReceipt()$ returns a sequence of event logs for some T; while $rootHash()$ denotes the function that computes the state root for some global state σ.

However, IoCs are not defined over big-step evaluations since this would imply a full transaction replay belonging to block β. We argue that this is beyond our scope since this would require redoing all the work carried out by EVM-level detection and rendering block-level IoCs useless. Therefore, beginning from a block β we want to investigate, we make use of ω computed during a big-step evaluation to lookup both the required $\langle BC, \sigma_0 \rangle$ and its corresponding $\langle BC \cdot \beta, \sigma_n \rangle$, as opposed to deriving the latter through full execution replay. The net effect is that of coarse-grained replay, where the side-effects at the blockchain and the BC and the global state (σ) are made available at block batches i.e. for $\langle BC, \sigma_0 \rangle$ and $\langle BC \cdot \beta, \sigma_n \rangle$ but not the interim σ_i, or any μ for that matter. To define the required partial replay, we make use of the Big-step lookup $\Updownarrow$:

Definition 3.19. For a given block β in blockchain *BC* containing n transactions, we define its associated Big-step lookup $\Updownarrow$ as:

$$\text{BIG-STEP-LOOKUP:} \frac{\begin{array}{c} \beta_{parent} \leftarrow \beta[parent] \\ \sigma_0 \leftarrow \omega[\beta_{parent}[stateRoot]] \\ \sigma_n \leftarrow \omega[\beta[stateRoot]] \end{array}}{\langle BC, \sigma_0 \rangle \Updownarrow \langle BC \cdot \beta, \sigma_n \rangle}$$

where β_{parent} represents β's parent as retrieved through its block hash stored in $\beta[parent]$; σ_0 and σ_n are the global states corresponding to β_{parent} and β respectively, with σ_n reflecting a global state update once β's n transactions get executed; both σ_0 and σ_n are retrieved, rather than computed, through the world map ω.

As for a common template, block-level IoCs are made up of expressions defined over historical σ_0/σ_n pairs, in turn, derived using the most recent ω and their corresponding βs. A block is flagged as containing an exploit transaction when all predicate expressions evaluate to true.

Definition 3.20. Block-level IoC template:

$$\text{Vulnerability:} \frac{\begin{array}{c} expression_1(\sigma_0, \sigma_n, \beta_{parent}, \beta) \\ expression_2(\sigma_0, \sigma_n, \beta_{parent}, \beta) \\ \dots \\ expression_n(\sigma_0, \sigma_n, \beta_{parent}, \beta) \end{array}}{\langle BC, \sigma_0, \rangle \Updownarrow \langle BC \cdot \beta, \sigma_n, \rangle}$$

where

- *Vulnerability*: Vulnerability class name to which the IoC belongs to

- β, β_{parent}: the block under investigation and its parent, respectively

- σ_0, σ_n: the pre/post global states

- $\langle BC, \sigma_0, \rangle$: the block-level pre-state comprising *BC* prior to β's addition and its corresponding global state σ_0

- $\langle BC \cdot \beta, \sigma_n, \rangle$: the block-level post-state reflecting the new global state σ_n resulting from when *beta* is appended to *BC* and its n transaction executed

- *expression_i*: operational semantics expressions defined over σ_0, σ_n, β_{parent}, β, with an IoC match occurring whenever all predicate expressions evaluate to true

3.2. IoCs for exploit transactions

In what follows, we use some vulnerable code snippets from [35] to illustrate how our IoCs are derived using IoC templates from Definitions 3.6 and 3.20.

Arithmetic over/underflow. Listing 3 is a code snippet with integer under/overflow vulnerabilities on lines 8 and 9, respectively, as a consequence of the absence of bounds checking in both cases.

Listing 3: Integer over/underflow in `Exchange.sol`.

```
1 mapping (address => uint256) public balanceOf;
2
3 // INSECURE
4 function transfer(address _to, uint256 _value) {
5     /* Check if sender has balance */
6     require(balanceOf[msg.sender] >= _value);
7     /* Add and subtract new balances */
8     balanceOf[msg.sender] -= _value;
9     balanceOf[_to] += _value;
10 }
```

The EVM-level IoC, shown in Definition 3.21, is defined over instruction class grouping all integer arithmetic operation $\otimes^{INT}$. While this IoC applies to any over/underflow IoC, the $TYPE$ parameter is contract-specific, identifying the lower/upper bounds of the integer type concerned. In the case of listing 3, we are concerned with an overflow in the value of the `balanceOf` mapping in the contract's storage, which is of type `uint256`. Therefore $TYPE$ in this case has to be instantiated as $\langle 0, 2^{256} - 1 \rangle$. Similarly, $VULN\_LOC$ is a place-holder for the instruction offsets related to the corresponding integer arithmetic opcodes. Any $\otimes^{INT}$ in this region is checked for an out-of-bounds condition, verifying whether $\rightsquigarrow$ results in an over/underflow. By first casting, the integer operation to its $\mathbb{Z}$ equivalent ($\odot^{INT}$), exploit verification reduces to checking whether $\odot^{INT}$'s evaluation falls outside of the $TYPE$'s range.

Definition 3.21. EVM-level integer over/underflow IoC.

$$\otimes_{TYPE}^{INT}: \frac{\begin{array}{c} pc \in VULN\_LOC \\ a',b' \in \mathbb{Z} \leftarrow \mathrm{cast}(a,b) \\ v' \leftarrow a' \odot^{INT} b' \\ v' < TYPE.min \vee v' > TYPE.max \\ v \in TYPE \leftarrow \mathrm{cast}(v') \end{array}}{\begin{array}{c} \langle\langle id, M, pc, l, a \cdot b \cdot s \rangle \cdot A, \sigma \rangle \\ \rightsquigarrow \langle\langle id, M, pc+1, l, v \cdot s \rangle \cdot A, \sigma \rangle \end{array}}$$

where:

- $\otimes^{INT} \overset{def}{=}$ `ADD | MUL | SUB | SDIV | ADDMOD | MULMOD | EXP` $TYPE.min/max$: define the lower/upper bounds for the EVM integer type of a given contract,

- $\odot^{INT}$ maps $\otimes^{INT}$ to its (infinite) $\mathbb{Z}$ counterpart, and

- $cast()$ maps arguments between EVM integer types and $\mathbb{Z}$.

Definition 3.22 describes the block-level IoC, with the out-of-bounds condition detectable at the contract storage level, rendering it contract-specific. In this case, the expressions are defined over the content of block β, and the global states prior and post its mining, σ_0 and σ_n respectively. The expressions constraining the coarsely replayed $\Downarrow$, represented by $\Updownarrow$, check for the existence of a T from β's TXs which: has `Exchange.sol`'s contract as its destination; compares the pre/post balances (*bal* and *bal′*) of the T's *sender*, and that of the account specified in the call data argument *_to*. Given that the `transfer` function is intended to shift funds from the *sender*'s to *_to* accounts, the side-effect of an over/underflow is one where either the prior's balance ends up increasing or vice-versa. When all these expressions evaluate to true, the block in question is flagged as containing an exploit transaction.

Definition 3.22. Block-level integer over/underflow IoC.

$$\text{o/u\_flow:} \frac{\begin{array}{c} TXs \leftarrow \beta[TXs] \\ \exists T \in TXs : \{ \\ \langle sender, to, value, data \rangle \leftarrow T \\ to == \gamma_{Exchange.sol} \\ bal \leftarrow \sigma_0[to]_s[balanceOf] \\ bal' \leftarrow \sigma_n[to]_s[balanceOf] \\ \_to \leftarrow T[data][to] \\ bal'[sender] > bal[sender] \vee bal'[\_to] < bal[\_to] \} \end{array}}{\langle BC, \sigma_0 \rangle \Updownarrow \langle BC \cdot \beta, \sigma_n \rangle}$$

where $\gamma_{Exchange.sol}$ denotes `Exchange.sol`'s contract account address

DoS with (unexpected) revert. Listing 4 is a highest-bidder type of contract, that

on receipt of a bid exceeding the current highest one, the contract-held funds are transferred back to the old bidder. Line 9, however, can be abused by an attacker making use of a destination contract account with a fallback payable function that always reverts. Line 9 therefore, always fails, pulling off a DoS on `Auction.sol`.

Listing 4: DoS vulnerability in `Auction.sol`.

```solidity
// INSECURE
contract Auction {
    address currentLeader;
    uint highestBid;

    function bid() payable {
        require(msg.value > highestBid);

        require(currentLeader.send(highestBid)); // Refund the old
    leader, if it fails then revert

        currentLeader = msg.sender;
        highestBid = msg.value;
    }
}
```

In this case, the EVM-level IoC in Definition 3.23 captures the small-step calculation ($\rightsquigarrow$) resulting in a failed `CALL` (returns 0), and which happens specifically at the vulnerable location. Temporarily, we ignore the possibility of a similar issue happening with a `DELEGATECALL`. By definition, we do not expect that a `STATICCALL` can ever result in a DoS of this type. No additional parameters apply in this case.

Definition 3.23. EVM-level DoS with (unexpected) revert IoC.

$$\otimes^{CALL}: \frac{\begin{array}{c} pc \in VULN\_LOC \\ r = 0 \end{array}}{\begin{array}{c} \langle\langle id, M, pc, l, s\rangle \cdot A, \sigma\rangle \\ \rightsquigarrow \langle\langle id, M, pc+1, l, r \cdot s\rangle \cdot A, \sigma\rangle \end{array}}$$

where $\otimes^{CALL} \stackrel{def}{=}$ `CALL`.

At the block level Definition 3.24 captures a Big-step lookup ($\Updownarrow$) where despite comprising a transaction value higher than the current highest bid, the highest bid remains the same. Once again, we note its contract-specific nature and that block-level flagging still applies.

Definition 3.24. Block-level DoS with (unexpected) revert IoC.

$$TXs \leftarrow \beta[TXs]$$
$$\exists T \in TXs : \{$$
$$\langle sender, to, value, data \rangle \leftarrow T$$
$$to == \gamma_{Auction.sol}$$
$$hbid \leftarrow \sigma_0[to]_s[highestBid]$$
$$hbid' \leftarrow \sigma_n[to]_s[highestBid]$$
$$\text{DoS:} \frac{value > hbid \wedge hbid' == hbid\}}{\langle BC, \sigma_0 \rangle \Updownarrow \langle BC \cdot \beta, \sigma_n \rangle}$$

where $\gamma_{Auction.sol}$: `Auction.sol`'s contract account address

Reentrancy. Back to `Exchange.sol`, Listing 5 this time shows a reentrancy issue in line 6. The contract wrongly assumes that line 8 will always be called prior to the same sender account attempting a further withdrawal. Yet, this is perfectly possible by a callback to `withdrawBalance()` performed by a malicious payable fallback function in the called contract.

Listing 5: Reentrancy vulnerability in `Exchange.sol`.

```
1 // INSECURE
2 mapping (address => uint) private userBalances;
3
4 function withdrawBalance() public {
5     uint amountToWithdraw = userBalances[msg.sender];
6     (bool success, ) = msg.sender.call.value(amountToWithdraw)(""); //
      At this point, the caller's code is executed, and can call
      withdrawBalance again
7     require(success);
8     userBalances[msg.sender] = 0;
9 }
```

The IoC reflects this exploit in Definition 3.25, where during a small-step evaluation that updates the contract storage (and therefore σ), the activation record stack A includes at least one entry; besides the one top, with the same *id* (the contract address). This IoC evaluating to true implies that when the sender's balance is being zeroed out (line 8), this step is not being performed by the outermost call but rather as a result of an internal transaction. This is exactly the scenario that the contract's programmer was wrongly assuming as not being possible.

Definition 3.25. EVM-level reentrancy IoC.

$$\otimes^{STORAGE} : \frac{\begin{array}{c} pc \in VULN\_LOC \\ \exists a \in A : a[id] = id \end{array}}{\begin{array}{c} \langle\langle id, M, pc, l, v \cdot s \rangle \cdot A, \sigma \rangle \\ \rightsquigarrow \langle\langle id, M, pc+1, l, s \rangle \cdot A, \sigma' \rangle \end{array}}$$

where $\otimes^{STORAGE} \overset{def}{=}$ `SSTORE`.

At the block level, the IoC in Definition 3.26 identifies the event of a withdrawal

transaction resulting in a σ_n where the contract's balance (*contractval'*) does not tally with the correct user balance withdrawal ($\neq contractval - user$), possibly even fully emptied.

Definition 3.26. Block-level reentrancy IoC.

$$TXs \leftarrow \beta[TXs]$$
$$\exists T \in TXs : \{$$
$$\langle sender, to, value, data \rangle \leftarrow T$$
$$to == \gamma_{Exchange.sol}$$
$$contractval \leftarrow \sigma_0[to][balance]$$
$$contractval' \leftarrow \sigma_n[to][balance]$$
$$user \leftarrow \sigma_0[to]_s[userBalances][sender]$$
$$\text{Reentr: } \frac{contractval' \neq (contractval - user)\}}{\langle BC, \sigma_0 \rangle \Updownarrow \langle BC \cdot \beta, \sigma_n \rangle}$$

where $\gamma_{Exchange.sol}$: `Exchange.sol`'s contract account address

---

4. The `EtherClue` tool

The EtherClue prototype is written in JavaScript, and its codebase is open-source[1]. It is intended to be connected either to a private Ethereum node or to a public cloud service. In either case, the nodes must be full archive nodes, meaning that they contain a full copy of mainnet, as they have to process all the code and transactions and not a fragment of them like, e.g. ArchiveNode[2]. In the case of EVM-level detection, the nodes must offer full transaction tracing capabilities, i.e. single-stepping over EVM instructions with full access to the EVM state. Its main components are described in the following paragraphs.

The **Orchestrator** loads and calls all the other components, coordinating investigation workflow execution. Given the address of a vulnerable contract, it firstly connects to a chosen Ethereum node via a *connector*. It then proceeds to obtain all the blocks falling within a particular range, as chosen by the investigator. It then invokes either a block or EVM-level *IoC detector*, to which it passes the list of block numbers (height) for exploit transaction detection using *IoC rules*. When invoking EVM-level detectors, it also takes care of passing any additional parameters, e.g. $TYPE.MIN/MAX$ for the integer over/underflow IoC (Definition 3.21).

Connectors integrate EtherClue with archive nodes. So far, we have developed one to integrate with any node exposing a JSON-RPC server[3] based on web3.js[4]. We also developed a connector that connects to Blockchair's API[5]. This connector is useful, for

---

[1] https://gitlab.com/simonjaquilina/etherclue

[2] http://archivenode.io

[3] https://geth.ethereum.org/docs/rpc/server

[4] https://web3js.readthedocs.io

[5] https://blockchair.com/api/docs

example, to retrieve all internal transactions associated with a specific external transaction. Using other types of nodes exposing their custom interface, say for an analytics service, e.g. Bloxy[6], entails writing a specific connector.

IoC detectors are tasked with fetching the relevant *IoC rules*, along with the relevant block data or transaction traces, for block and EVM-level IoCs, respectively, along with their corresponding account states to carry out exploit transaction detection. The EVM-level detector provides two modes of operation: either i) a *local mode* involving trace retrieval followed by local detection; or ii) a *custom tracer mode* that avoids transaction retrieval by implementing detection as a `tracer` JavaScript object, accepted by the `debug_traceTransaction` RPC endpoint and which is executed remotely on the node itself. Both detectors can also operate in a *cached mode* by retaining all previously fetched blocks/traces.

Call context tracker. This component is used by the EVM-level IoC detector. Whenever a message call-generating instruction is encountered, namely: `CALL`, `STATICCALL`, `CREATE`, or `CREATE2`; it ensures that IoC detection is disabled until a `RETURN` or `REVERT` is encountered, as otherwise this would confuse those expressions defined over code locations.

IoC rules are JavaScript objects called by detectors per block/trace. The block-level objects' prototype (or "class") must implement the method `consider`, providing the encoding of IoC semantics, which is called for each block retrieved by its corresponding IoC detector. It exposes the following interface: `function(blockNumber, txs, accounts, blockBefore, blockAfter);` where the parameters stand for the block number, its vulnerability-relevant transactions, the accounts concerned, and the pre/post account states. The EVM-level objects are called per transaction trace. Their `consider` method is exposed as: `function(address, tx, trace, callback);` where parameters stand for the contract address, transaction hash and its trace, along with callback function for reporting alert details respectively. For both object types, a number of helper properties and methods are also needed[7].

5. Evaluation

A number of diverse experiments were carried out to provide further insight into EtherClue's IoC effectiveness and performance and are presented in the following paragraphs.

5.1. Experimental Setup

First, we employed a mix of synthetic contracts embedding common vulnerabilities and smart contracts from Ethereum mainnet. In this manner, we strike a balance between control over blockchain content and realism. The former provides a means to maximise variations and therefore provides for a more comprehensive insight into

---

[6]https://bloxy.info/

[7]See example objects for further details at: https://gitlab.com/simonjaquilina/etherclue/-/tree/master/src/block-level-ioc-d/ioc-detectors,https://gitlab.com/simonjaquilina/etherclue/-/tree/master/src/evm-level-ioc-d/ioc-detectors

EtherClue. The latter provides an opportunity to demonstrate practical deployment. In the former setup, we connect EtherClue to a private `geth 1.9.21-stable` node whose blockchain is populated with multiple benign and exploit transactions. Both `geth` and EtherClue are hosted on an Intel Core i5 1.3 GHz with 4GB of RAM machine. For the mainnet setup, we connected EtherClue to both Etherscan and ArchiveNode to obtain the full internal transaction list for each external transaction of interest and the EVM traces per external/internal transaction, respectively.

The datasets[8] used for experimentation have been publicly released to facilitate further experimentation with Ethereum forensics. It includes a private blockchain, aiming to emulate real vulnerable smart contracts and exploit transactions, but purposely aiming to maximise content variation, along with labelled (manually verified) attack transactions for the Ethereum mainnet case studies.

The synthetic smart contracts are: *Underflow* - `TargetUnderflow` and `DelayedUnderflow` that include underflow vulnerabilities that directly or indirectly (i.e. overflow impact on control-flow) affect contract storage respectively; *Overflow* - `SimulationBECToken` which embeds the BECToken overflow vulnerability[9]; *Reentrancy* - `Bank` and `ProductVote` that insecurely allow recursive Ether withdrawal and multiple-voting at the cost of a single vote respectively; *DoS* - `SimulationKotET` which embeds the King-of-the-Ether-Throne (KotET)[10] DoS vulnerability. The mainnet contracts are: BECToken (Overflow), KotET (Dos), The DAO[11] (Reentrancy) and SpankPay[12] (Reentrancy).

5.2. Results

Exploit transaction detection. Table 2 illustrates the detection results of exploit transactions targeting the synthetic contracts at both the EVM and block levels. For each vulnerability, a session of transactions was submitted, consisting of both benign and exploit transactions. Benign transactions comprise vulnerable smart contract deployment and subsequently bring them to a state that renders them appealing for exploitation, e.g. setting up products and votes in `ProductVote` or the highest bidder in `SimulationKotET`. Further transactions are sent to the server as background noise in parallel to exploit transactions, enabling a realistic block mining process. The case of `TargetUnderflow` is quite particular. It represents the case where an underflow results in a storage variable that is propagated to other variables in the same storage, with all transactions ending up being exploits. As such, the benign transaction count varies per smart contract.

Notably, no false positives were registered in either case, while the EVM-level returned fully accurate detection in all cases. Nevertheless, the block-level missed out on four malicious transactions targeting `SimulationBECToken`. Three of them were missed since the transactions were internal ones, and their information is not

---

[8]`https://gitlab.com/simonjaquilina/etherclue/datasets`
[9]`https://peckshield.medium.com/alert-new-batchoverflow-bug-in-multiple-erc20-smart-contracts-cve-2018-10299-511067db6536`
[10]`https://www.kingoftheether.com/postmortem.html`
[11]`https://www.coindesk.com/understanding-dao-hack-journalists`
[12]`https://medium.com/spankchain/we-got-spanked-what-we-know-so-far-d5ed3a0f38fe`

Contract Account	TXs No.	TP EVM	FP EVM	FN EVM	Blocks No.	TP Block	FP Block	FN Block
Bank★	82	6	0	0	42	3	0	0
DelayedUnderflow†	23	11	0	0	22	11	0	0
ProductVote★	115	20	0	0	38	15	0	0
SimulationBECToken†	56	12	0	0	25	4	0	4
SimulationKotET‡	151	4	0	0	24	4	0	0
TargetUnderflow†	20	20	0	0	20	20	0	0

Table 2: Exploit transaction detection at the EVM and Block-level IoCs.
†-Over/underflow ‡-DoS ★-Reentrancy

retained at the block level (i.e. by β) that is investigated. The remaining transaction was missed since the affected storage variable was overridden by a subsequent update in the same block, occluding the attack side-effect in question. It is noteworthy to mention that these limitations were exposed by our approach aiming to maximise variation of blockchain content rather than limiting experimentation solely to currently available mainnet transactions.

Performance. We measured EtherClue's performance in terms of investigation workflow duration for gradual increases in contract size. The contract size varies in terms of instruction counts, bound to affect EVM-level IoC detectors due to an increase in small-steps, and storage size, bound to affect block-level IoC detectors when fetching contract storage. For the former, we modified DelayedUnderflow.sol with additional ADD instructions (serving as a *NOP*) until we reached the 2 million gas unit utilization mark, far exceeding typical transaction trace size (including the internal calls). For the latter, we took a similar approach with TargetUnderflow.sol, this time by making use of an mapping(uint=>uint) storage variable, conveniently allowing one of its functions to add a key/value pair at each invocation. In both cases, we do not factor in the network latency portions as part of the processing times since they are the same for both the EVM and block-level IoC detectors. Moreover, we did not apply any parallelisation or further optimisations in EtherClue's implementation, leaving this task for future research. In fact, the lack of parallelisation, illustrates the efficacy of our proposed solution since the results can be linearly improved, in terms of time, using more cores.

Figure 5 (upper) shows how an increase in instruction count, as expected, only affects the EVM-level. However, the slow-down can be mitigated by the custom tracer mode. Whenever cached entries can be used, the slow-down can be mitigated even further. In any case, the increase in processing time is linear. On the other hand, Block-level IoCs are slowed down by an increase in contract storage size, as shown in Figure 5 (lower). In this case, we did not bother with the custom tracer mode since the processing times for local modes remained constant. Once again, results demonstrate that caching can drastically decrease processing time while the slow-down is linear. Contract size varied in terms of instruction counts, bound to affect EVM-level IoC detectors due to an increase in small-steps and storage size, bound to affect block-level IoC detectors when fetching contract storage. For the former, we modified DelayedUnderflow.sol with additional ADD instructions (serving as a *NOP*) until we reached the 2 million gas unit utilization mark, far exceeding typical transaction

trace size (including the internal calls). For the latter, we took a similar approach with `TargetUnderflow.sol`, this time by making use of an `mapping(uint=>uint)` storage variable, conveniently allowing one of its functions to add a key/value pair at each invocation. In both cases, we do not factor in the network latency portions as part of the processing time, since they are the same for both the EVM and block-level IoC detectors. Moreover, we did not apply any parallelisation or further optimisations in EtherClue's implementation, leaving this task for future research.

Practicality. Table 3 shows detection/duration details of the four mainnet case studies (column 1) conducted over ArchiveNode using the EVM-level IoC detector in custom tracer mode. Based on the result attained using the synthetic accounts, this configuration provides the best option for effectiveness and performance. The EVM-level IoC detector used the Blockchair connector to retrieve all internal transactions resulting from external ones that belong to the chosen block range. Columns 2-4 show the attack day that was considered, the first day of the attack in each case, followed by the block ranges and their corresponding number of blocks. In all cases, these comprise all the blocks mined on that day. The fifth column shows the full external transaction count included in that block range. Column 6 shows the number of those (external) transactions actually considered to be relevant by the IoC detector. For BECToken, given that the vulnerability is located in the `batchTransfer()` public function, which is never called internally by the smart contract itself, these are taken to be all external transactions calling `batchTransfer()` directly, or indirectly via an internal transaction. In the case of KotET, all transactions targeting its address are considered, since these could all result in an internal call that reverts unexpectedly. For TheDAO and SpankPay, the IoC detector considers all external transactions, targeting any smart contract address, with at least one internal transaction targeting their own address.

The results confirm the efficacy of EVM-level in the case of synthetic contracts. True (TP) and false positives (FP) were computed through manual verification of all flagged transactions by using EtherScan[13]. In the case of BECtoken, only one transaction was flagged, coinciding with the transaction hash reported in its corresponding report. For KotET, we checked that the flagged transactions had errors (reverted with an out-of-gas exception) but still completed. As for the TheDao and SpankPay we checked that the detected transactions had multiple internal message calls to TheDarkDAO (attacker's source account). Similarly to related work [21], in the lack of appropriate ground truth, false negatives could no be accounted for in a precise manner. However, for TheDAO and SpankPay we did confirm that EtherClue was able to detect all exploit transactions detected by Sereum [22]. Duration times include the entire period as of when the EtherClue orchestrator is invoked with the given block range until all transaction processing terminate. Even though EVM-level processing is the most expensive, results show that the investigation duration times remain practical throughout the case studies. As already discussed, the time taken to investigate a single transaction depends on contract sizes, both in terms of instruction counts and storage sizes (see Figure 5).

Moreover, the total investigation time is impacted by the total number of transactions considered relevant for the case in hand. Taking the case of the TheDAO, which is

---

[13]https://etherscan.io/

Contract	Date	Block Range	Blocks No.	TX Full	TX Relevant	TP	FP	Duration (s)
BECToken[†]	22/04/18	5482796-5488697	5901	748858	17	1	0	113
KotET[‡]	07/02/16	964910-969996	5086	14264	9	2	0	3
TheDAO*	17/06/16	1717562-1723410	5848	66558	5666	448	0	1838
SpankPay*	07/10/18	6467090-6473330	6240	497767	201	7	0	172

Table 3: Ethereum mainnet case studies (EVM-level detection).
†-Overflow ‡-DoS ⋆-Reentrancy

by far the most expensive among the four case studies, a non-parallel implementation of EtherClue can process a block in 0.3s. This is already faster than the rate at which a new block is currently mined in mainnet, falling in the 10-20s range[14], and therefore implying that EtherClue can even be deployed as a real-time detector of exploit transactions.

Comparison with state of the art. As described in Section 1, EtherClue's scope complements smart contract-centric vulnerability detection tools with a transaction-centric one intended for incident response. In this respect, TxSpector [21], Sereum [22] and DEFIER [23] fall into this category, presenting candidate comparison targets. However, one of the main issues of this research topic is that it lacks a standardised framework to leverage a sound comparison between the state of the art methods. The main implications, also mentioned in the literature [21], are: the lack of comprehensive ground truth beyond the transaction hashes detected and manually followed up by individual efforts and the availability of openly available rules/analysis routines.

Nevertheless, given that TxSpector's rulesets are openly available, while the transactions detected by Sereum are available in a properly labelled way, we are in a position to compare EtherClue's EVM-level IoCs with TxSpector based on ruleset definition and to cross-check the detected reentrancy exploit transactions with Sereum's. No form of comparisons with DEFIER is yet possible since, firstly, DEFIER requires a set of seed transactions to be made available, something that EtherClue does not assume, while the notion of an attack transaction in DEFIER goes beyond strictly that of exploit transactions to also include the entire attack kill-chain. Moreover, block-level IoC comparison is not possible since none of the existing tools provide it. We expand further on the requirements for a standardised framework for sound and fair comparison in the ensuing discussion (Section 5.3).

We start by comparing ruleset definition between EtherClue and TxSpector. Therefore, for EtherClue we consider its IoC rules and for TxSpector, its detection rules. Taking the *DoS with (unexpected) revert* as a case study, Listing 6 shows TxSpector's corresponding detection rule. Its detection engine is implemented using the `Soufflé` high-performance Datalog query tool. Control and data flow dependencies, such as opcode sequences and information flow, are extracted from transactions traces and populate a knowledge base, in turn, queried using relational logic rules. This declarative approach is a strong point for TxSpector; however, it is clear from this example that this tool is intended primarily for users knowledgeable of program analysis. Ultimately,

---

[14]see https://etherscan.io/blocks

this is the same approach used for static code analysis but adapted to work on runtime traces. In Listing 6, line 7 extracts all `CALL` opcodes, while line 8 checks whether their return values are eventually used by a `JUMPI` opcode (a conditional jump). Yet, the `JUMPI` opcode may not immediately follow its corresponding `CALL`, nor act directly on the returned value itself. Therefore, the rule on lines 1-4 is required to capture this additional complication, using further control/data flow dependencies.

Listing 6: TxSpector detection rule for DoS.

```
1 jumpiDep(jumpiIdx, jumpiDepth, depIdx, depVal) :-
2     op_JUMPI(_, _, jumpiCond, jumpiIdx, jumpiDepth, _),
3     jumpiIdx > depIdx,
4     depends(jumpiCond, depVal).
5
6 UncheckedCall(args) :-
7     op_CALL(_, _, _, callRet, callIdx, 1, _),
8     !jumpiDep(jumpiIdx, 1, callIdx, callRet).
```

In EtherClue's case, given IoCs are defined as attack side-effects, it only requires reasoning about individual EVM-level steps. Furthermore, EtherClue is meant to complement vulnerability detectors and so can avail from known vulnerability locations inside the compiled EVM code. Starting with Definition 3.23, the IoC rule verifies whether: i) the current opcode is a `CALL`, ii) it falls within the vulnerability location, and iii) the return value is 0. These three checks are simpler and more elegant than TxSpector's detection rule (Listing 6), which is burdened with ascertaining whether the EVM trace being analyzed corresponds to a vulnerability. Therefore, EtherClue's IoC rules do not have to follow the complex control/data flow dependency analysis which impedes TxSpector's rules.

Notably, the IoC's implementation is not restricted to a specific programming paradigm either. The current EtherClue prototype implements IoC rules in JavaScript, an example fragment of which is shown in Listing 7. `arStack` on line 2 is an instance of the *call context tracker* described in Section 4, and which is used to step through the transaction trace while keeping track of call-depths and activation records corresponding to internal transactions. Recall from Definition 3.2 that the activation record includes all runtime information, encompassing stack/memory content, EVM code and the program counter. `arStack` is therefore central for the IoC rule verification (i.e., the three previously stated checks related to Definition 3.23) performed in lines 3, 7 and 9. Finally, the `callback()` in line 10 flags the transaction as an exploit. This occurs whenever all checks succeed as per the IoC template in Definition 3.6.

Listing 7: EtherClue IoC rule for DoS.

```
1 ...
2   var previousStep = arStack.active().nthStep(1);
3   if(previousStep.op === Op.CALL){
4     var stackAfter = step.stack;
5     var r = BigInt(stackAfter[stackAfter.length - 1], 16);
6     // Check if the result is an IoC for DoS.
7     if(r.eq(0)){
8       trace.structLogs.forEach(function(step2, i){
9         if(step2.pc == 940 || step2.pc == 941){
10            callback(arStack.active().id, tx, previousStep, index, ""
     );
```

```
11          }
12        });
13      }
14    }
15  ...
```

The main observation here is that it should be simpler to implement a series of checks on the pre/post states of EVM instructions rather than defining vulnerabilities using program analysis. Furthermore, EtherClue's IoCs can also be used at the block level, something that TxSpector does not provide. We elaborate further on this point in the upcoming discussion.

Another important aspect that prevents a fair comparison is that, for instance, TxSpector discards transactions not processed according to a timeout threshold. Therefore, the sample set may vary when comparing with other tools such as EtherClue, since the latter analyses all the transactions and is not restricted by performance thresholds. While we believe that the detection tools' performance is relevant, enabling a timeout requires further discussion since individuals may not care about the time analysis if their losses are enormous.

Finally, we further elaborate on the outcomes of TheDAO experiment. According to the original reports, Sereum flagged 2112 transactions whilst TxSpector flagged 2008. EtherClue was able to flag 2208, since it identified 96 exploit transactions that ended up with an exception, and therefore failed to update the blockchain state. According to Sereum documentation, it does not consider non-successful exploit transactions, yet during our analysis, we found out that 7 of such transactions were included in their outcomes, probably due to a bug/error. The latter means that EtherClue achieved the same performance as Sereum (i.e., a total of 2015 transactions if we remove the non-successful ones), which is a tool focused on this specific exploit.

5.3. Discussion

Our research and experimental findings provide some rather interesting insights about blockchain forensics and the corresponding IoCs. In the following paragraphs we discuss their main points.

EVM-level IoCs provide the best option. EVM-level IoCs carry a considerable larger computational load. Nonetheless, one should take into consideration that such IoCs can generalise over an entire exploit class. Moreover, they can specifically flag exploit transactions within their block, and their effectiveness cannot undermined neither by the 'unavailable internal transactions' nor by the 'occluded side-effect' issues of the block level. Therefore, EVM-level IoC-centric investigations can be more effective. While such IoCs impose a much larger computational load, our extended experiments illustrate that the cost is tolerable and applicable in real-world scenarios. Beyond a pre-processing step involving transaction tracing, a larger number of analysis steps is also required due to the small-step execution semantics. Yet, the mainnet case studies show that processing times remain practical even for transactions that consume the largest possible amount of gas.

Block-level IoCs can still be of value. During investigations in which response times are critical, the lightweight nature of IoCs at this level may still render them useful in the form of an 'approximate' triage-pass. Albeit not fully accurate, this pass can still

31

help with the quick identification of a sub-set of exploit transactions, especially given that IoCs at this level err on false negatives rather than false positives.

Automated IoC definition/code generation. The potential value of a triage-pass based on block-level IoCs, however, is undermined by the fact that they are contract-specific, and IoC definition followed by IoC Rule implementation can be time-consuming. Yet, the formal semantics upon which IoCs are defined can provide a solid basis for automation. More precisely, the symbolic execution of a sample exploit transaction could provide the basis for automation by generating all the states associated with successful exploitation and from which to derive IoCs directly in code form. Given that symbolic execution does not require concrete inputs, this enables the generation of IoCs in a generic manner through symbolic inputs. Furthermore, symbolic execution engines for the EVM, e.g. Manticore [36], are already available.

A standardised framework for comparison. Ultimately, given the limited way we were able to compare EtherClue with existing tools so far, further enhancements require the support of clear comparisons for proper evaluation. Specifically, we encountered substantial difficulties in making use of datasets, at least beyond the limited one provided by Sereum [22], due to the lack of properly labelled ground truth and harmonized definitions of what constitutes an exploit transaction, amongst others. Therefore, we believe that studying standardised criteria to measure the quality of the tools with unified benchmarks and metrics is a promising research line in its own right. From our own experience, we propose that such a framework should at least cover: i) Tool categorisation in terms of which modes they can operate in, i.e. contract, block, transaction, or individual EVM instruction levels; ii) In the case of tools supporting a contract-level operation mode, whether they require the source code or whether they can work with just the decompiled bytecode; iii) Make the ruleset/analysis routines available for comparison, at least in object form, but more importantly highlighting the exact scope of their detection; iv) A ground-truth dataset that uses different labels for attack transactions at the different stages of the attack kill-chains, specifically identifying exploit transactions as only those that directly exercise a security bug; v) Finally, enhance all exploit transaction labels with related block-level, internal message call and instruction-level annotations.

6. Conclusions

In this work, we showed that IoCs defined over the side-effects of Ethereum smart contract execution is an effective way to identify exploit transactions. The primary contribution is EtherClue, a DFIR tool that complements vulnerability detectors by post-factum exploit transaction detection. A model that abstracts smart contract execution underpins the method for defining IoCs, and which in turn builds upon a formalism; previously used for symbolic smart contract execution, to fit the needs of digital forensics. Experimentation with common exploit classes demonstrates the effectiveness and real-world practicality of EtherClue compared with other state-of-the-art approaches. Results point towards defining IoCs at the EVM level, rather than the block level since IoCs can generalise over exploit classes and achieve better precision.

Future work will focus on defining further IoCs for the ever-growing list of exploit classes, along with further experimentation on mainnet. This effort will, however,

firstly require a parallel implementation of EtherClue IoC detectors. This optimised implementation will also be useful to experiment further with the idea of applying EtherClue real-time detection, and perhaps even for early detection inside transaction pools. Deriving IoCs automatically from exploit samples is also a possibility given EtherClue's operational semantics foundation. Finally, by integrating EtherClue with data mining tools, e.g. DEFIER, and blockchain analytics, e.g. Bloxy, it will be possible to automate the process of forensic timeline analysis, providing investigators with a complete insight into an incident with minimal manual effort. Moreover, we argue that a standardised framework for tool comparison can be a catalyst for advancing the state of the art in Ethereum forensics and blockchain technologies in general.

Acknowledgements

References

[1] N. Atzei, M. Bartoletti, T. Cimoli, A survey of attacks on Ethereum smart contracts (SoK), in: International conference on principles of security and trust, Springer, 2017, pp. 164–186.

[2] H. Chen, M. Pendleton, L. Njilla, S. Xu, A survey on Ethereum systems security: Vulnerabilities, attacks, and defenses, ACM Computing Surveys (CSUR) 53 (3) (2020) 1–43.

[3] D. Branch, dApps: Beyond germs transmitted by handshakes, `https://blog.securityevaluators.com/dapps-beyond-germs-transmitted-by-handshakes-edb48b468922`, accessed: 2021-01-12 (2019).

[4] G. Prisco, Understanding the DAO attack, `https://bitcoinmagazine.com/articles/the-dao-raises-more-than-million-in-world-s-largest-crowdfunding-to-date-1463422191`, accessed: 2021-01-12 (2016).

[5] K. O'hara, Smart contracts-dumb idea, IEEE Internet Computing 21 (2) (2017) 97–101.

[6] Y. Smaragdakis, Ethereum Pawn Stars: "$5.7M in hard assets? Best I can do is $2.3M", `https://medium.com/dedaub/ethereum-pawn-stars-5-7m-in-hard-assets-best-i-can-do-is-2-3m-b93604be503e`, accessed: 2021-01-12 (2021).

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

[8] N. Grech, L. Brent, B. Scholz, Y. Smaragdakis, Gigahorse: thorough, declarative decompilation of smart contracts, in: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE, 2019, pp. 1176–1186.

[9] G. Grieco, W. Song, A. Cygan, J. Feist, A. Groce, Echidna: effective, usable, and fast fuzzing for smart contracts, in: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2020, pp. 557–560.

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

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

[12] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, B. Roscoe, Reguard: finding reentrancy bugs in smart contracts, in: 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion), IEEE, 2018, pp. 65–68.

[13] D.-Y. Kao, S.-C. Hsiao, The dynamic analysis of wannacry ransomware, in: 2018 20th International Conference on Advanced Communication Technology (ICACT), IEEE, 2018, pp. 159–166.

[14] G. Wood, et al., Ethereum: A secure decentralised generalised transaction ledger, Ethereum project yellow paper 151 (2014) (2014) 1–32.

[15] V. Buterin, Etheum white paper, https://ethereum.org/en/whitepaper/, accessed: 2021-01-12 (2020).

[16] Ethereum, Solidity, https://docs.soliditylang.org/en/latest, accessed: 2021-01-12 (2020).

[17] Go-Ethereum, EVM tracing, https://geth.ethereum.org/docs/dapp/tracing, accessed: 2021-01-12 (2020).

[18] T. Cook, A. Latham, J. H. Lee, DappGuard: Active monitoring and defense for solidity smart contracts, Retrieved July 18 (2017) 2018.

[19] S. Farrugia, J. Ellul, G. Azzopardi, Detection of illicit accounts over the Ethereum blockchain, Expert Systems with Applications 150 (2020) 113318.

[20] N. Kumar, A. Singh, A. Handa, S. K. Shukla, Detecting malicious accounts on the Ethereum blockchain with supervised learning, in: International Symposium on Cyber Security Cryptography and Machine Learning, Springer, 2020, pp. 94–109.

[21] M. Zhang, X. Zhang, Y. Zhang, Z. Lin, TXPTECTOR: Uncovering attacks in Ethereum from transactions, in: 29th USENIX Security Symposium (USENIX Security 20), 2020, pp. 2775–2792.

[22] M. Rodler, W. Li, G. O. Karame, L. Davi, Sereum: Protecting existing smart contracts against re-entrancy attacks, arXiv preprint arXiv:1812.05934 (2018).

[23] L. Su, X. Shen, X. Du, X. Liao, X. Wang, L. Xing, B. Liu, Evil under the sun: Understanding and discovering attacks on ethereum decentralized applications, in: 30th USENIX Security Symposium (USENIX Security 21), USENIX Association, Vancouver, B.C., 2021.
URL https://www.usenix.org/conference/usenixsecurity21/presentation/su

[24] S. Bragagnolo, M. Marra, G. Polito, E. G. Boix, Towards scalable blockchain analysis, in: 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), IEEE, 2019, pp. 1–7.

[25] L. Wu, S. Wu, Y. Zhou, R. Li, Z. Wang, X. Luo, C. Wang, K. Ren, EthScope: A transaction-centric security analytics framework to detect malicious smart contracts on Ethereum, arXiv preprint arXiv:2005.08278 (2020).

[26] M. Wohrer, U. Zdun, Smart contracts: security patterns in the Ethereum ecosystem and solidity, in: 2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE), IEEE, 2018, pp. 2–8.

[27] ConsenSys, Ethereum smart contract best practices, https://consensys.github.io/smart-contract-best-practices/general_philosophy/, accessed: 2021-01-12 (2021).

[28] M. Rodler, W. Li, G. O. Karame, L. Davi, Evmpatch: Timely and automated patching of ethereum smart contracts, arXiv preprint arXiv:2010.00341 (2020).

[29] Z. Cheng, X. Hou, R. Li, Y. Zhou, X. Luo, J. Li, K. Ren, Towards a first step to understand the cryptocurrency stealing attack on Ethereum, in: 22nd international Symposium on research in Attacks, Intrusions and Defenses ({RAID} 2019), 2019, pp. 47–60.

[30] C. Colombo, J. Ellul, G. J. Pace, Contracts over smart contracts: Recovering from violations dynamically, in: International Symposium on Leveraging Applications of Formal Methods, Springer, 2018, pp. 300–315.

[31] M. Weber, G. Domeniconi, J. Chen, D. K. I. Weidele, C. Bellei, T. Robinson, C. E. Leiserson, Anti-money laundering in bitcoin: Experimenting with graph convolutional networks for financial forensics, arXiv preprint arXiv:1908.02591 (2019).

[32] L. Zarpala, F. Casino, A blockchain-based forensic model for financial crime investigation: The embezzlement scenario, arXiv preprint arXiv:2008.07958 (2020).

[33] A. H. Lone, R. N. Mir, Forensic-chain: Blockchain based digital forensics chain of custody with PoC in Hyperledger composer, Digital Investigation 28 (2019) 44–55.

[34] L. Ahmad, S. Khanji, F. Iqbal, F. Kamoun, Blockchain-based chain of custody: towards real-time tamper-proof evidence management, in: Proceedings of the 15th International Conference on Availability, Reliability and Security, 2020, pp. 1–8.

[35] ConsenSys, Ethereum smart contract best practices: Known attacks, `https://consensys.github.io/smart-contract-best-practices/known_attacks/`, accessed: 2021-01-12 (2020).

[36] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, A. Dinaburg, Manticore: A user-friendly symbolic execution framework for binaries and smart contracts, in: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2019, pp. 1186–1189.
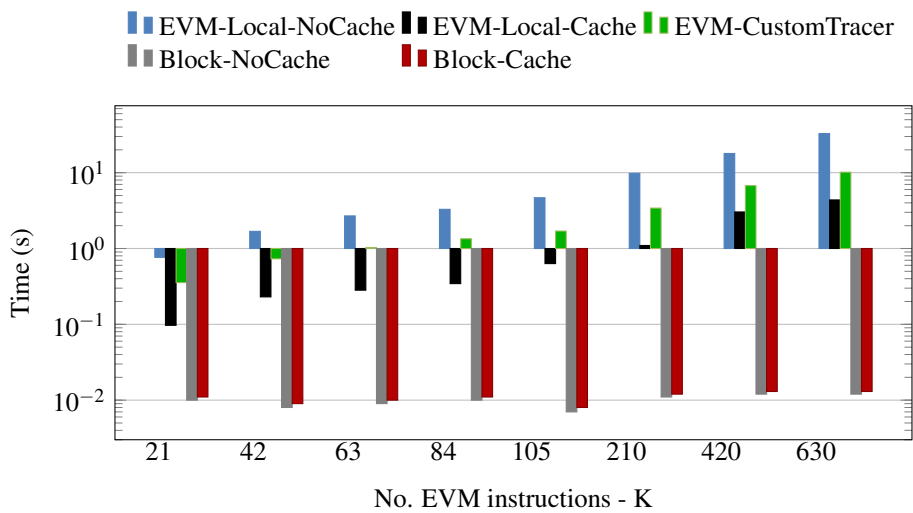
(a) Performance analysis for a gradual increase in contract size in terms of instructions.
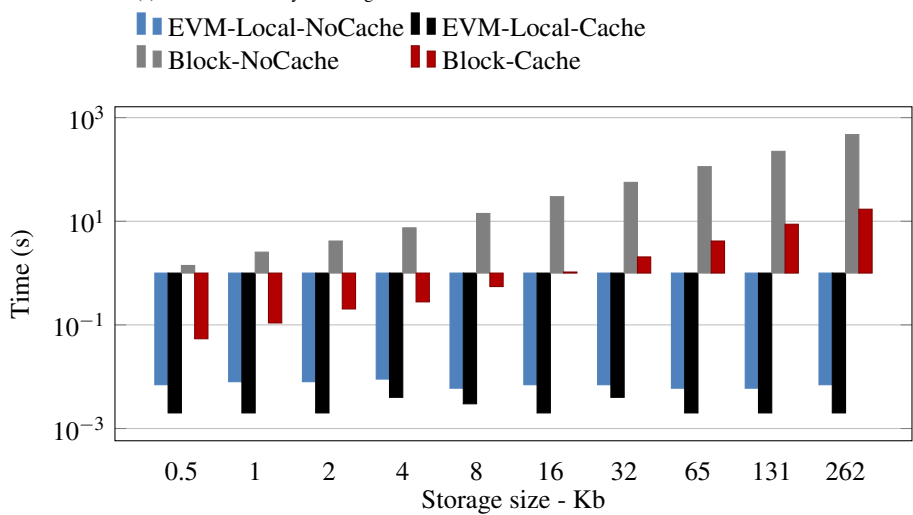
(b) Performance analysis for a gradual increase in contract size in terms of storage

Figure 5: Performance analysis for a gradual increase in contract size.