EtherProv: provenance-aware detection, analysis, and mitigation of Ethereum smart contract security issues

Shlomi Linoy
Faculty of Computer Science
University of New Brunswick
Fredericton, Canada
slinoy@unb.ca

Suprio Ray
Faculty of Computer Science
University of New Brunswick
Fredericton, Canada
sray@unb.ca

Natalia Stakhanova
Department of Computer Science
University of Saskatchewan
Saskatoon, Canada
natalia@cs.usask.ca

***Abstract*—The rapid adoption of blockchain technologies and particularly smart contracts has been overshadowed by numerous security concerns. Over the past few years, a number of reports exposed smart contracts vulnerabilities and exploits, which mainly stem from the immaturity of the field, and consequently a lack of knowledge and tools for automated analysis and verification of smart contracts. The restricting properties of the blockchain environment, such as the immutability of deployed contracts, encumber the analysis and mitigation of vulnerabilities and bugs in deployed contracts. To address these challenges, we propose EtherProv, a novel provenance tracking system that leverages static and dynamic analysis synergy to enable detection and mitigation of known security issues in Ethereum smart contracts. EtherProv leverages Solidity source code static and dynamic analysis data through contract bytecode instrumentation. The collected data is transformed into a unified, high-level representation, which can be queried using concise and descriptive Datalog queries. Within the provenance framework, EtherProv is able to analyze contracts' execution flow over time, to detect vulnerabilities within a single contract execution flow and across multiple interacting contracts, and to mitigate new security threats in already *deployed* contracts. Our evaluation shows that EtherProv can efficiently and precisely identify vulnerable contracts with an average contract instrumentation gas overhead of 18.9%.**

***Index Terms*—Blockchain, security, provenance, smart contracts, vulnerability-analysis, debugging**

I. Introduction

Blockchain technologies enable different parties who do not trust each other to share information using a robust consensus protocol, which eliminates the need for a central authority. There are over 50 blockchain platforms available in the world today [1]. Many of them enable the creation and automated execution of smart contracts, which represent auditable and enforceable agreements, in a decentralized way. Among them are Ethereum [2] (the first platform to support smart contracts), Tezos [3], EOS [4], Cardano [5], and Hyperledger Fabric [6].

A *smart contract* (abbreviated as *contract*) is a piece of computer code that enables users to create their own arbitrary rules for ownership and state transition functions. The contract is written in a high-level language such as Solidity and is compiled into bytecode, e.g., EVM (Ethereum Virtual Machine) bytecode in the Ethereum platform. With the wide adoption of contracts, their security emerged as a critical concern. As any software code, contracts are susceptible to security vulnerabilities and exploits. Due to the immaturity of the field, the contract developers often lack security knowledge to ensure a contract's secure execution, thus the burden of security verification falls solely on the existing tools and platforms.

The majority of the current security verification approaches that address *known* security issues leverage static analysis of source code [7]–[10] or bytecode [11], [12]. Analysis of the Solidity source code provides substantial benefits over bytecode analysis, as a large amount of information is lost in the compilation process. Bytecode analysis incurs imprecise resolution of memory and storage offsets when identifying variables [11], and jump targets when identifying stack locations statically [12], which may lead to higher false positive/negative rates. The static analysis of contracts primarily aims to discover the presence of potential code security vulnerabilities. Understanding whether these vulnerabilities are exploitable requires an analysis of the contracts' execution flow, and hence, execution of the contracts' bytecode. Contracts can be called by external accounts, as well as by other contracts, therefore dynamic parameters (e.g., function call parameters, current storage states), externally called contracts and libraries, and transaction data, should all be taken into account while evaluating the vulnerability and exploitability of the contracts.

Current approaches for addressing *new* security issues enable the analysis of deployed contracts by modifying a client node with considerable instrumentation [13]–[15], or by replaying historic transactions [16]. While these approaches enable a fine grained data collection and analysis at the function/instruction level, they incur considerable computation and storage resources and are thus often not suitable for practical deployment on a production blockchain.

Due to the immutability of deployed contracts, handling unaddressed security flaws or malicious behavior is challenging. For example, in 2016 an exception handling bug, which was discovered in the popular contract game "King of the

Ether Throne" resulted in gas transfer failures. To resolve the issue, the developers were forced to manually reimburse some players and publicly request players to not use their contract [17] in order to prevent further monetary loss.

As the rate of new security attacks increases, there is a need for an environment that can facilitate an efficient analysis and mitigation of deployed contracts' security issues to reduce damage and monetary loss in a timely manner. To achieve this, it is necessary to collect fine-grained contract execution trace, which we call smart contract execution flow provenance or execution flow provenance for short.

To address these challenges, we propose **EtherProv**, a smart contract execution provenance tracking system that leverages static and dynamic analysis synergy *to efficiently detect known security issues, analyze new security issues, and mitigate unaddressed security issues across interacting Ethereum contracts, and across transaction history*.

EtherProv makes use of contract bytecode instrumentation, which as opposed to the existing approaches, does not require client node modification or transactions replay. To support the analysis of contract security issues, EtherProv collects code information through static analysis and enables tracing of an execution flow spanning multiple interacting deployed contracts. This is achieved through efficient control flow graph (CFG) instrumentation. When a deployed contract is executed, the execution flow path encoding is emitted to the blockchain, which can then be retrieved and decoded to the full executed path. To enable a more thorough analysis, EtherProv collects on-line dynamic data, e.g., block/transaction ids, caller details, function call parameters, storage state values, etc. The collected dynamic data and encoded paths are then transformed to a unified representation and stored in a provenance database as Datalog facts.

Security analysis in EtherProv is based on the intuition that while security concerns in contracts may be complex, once these issues are discovered and defined, they can be detected by checking a contract's compliance with the known security properties. To detect and analyze known security issues within a single contract and across multiple contracts, the provenance database is queried using *compliance queries* written in Datalog language.

Besides identifying known security issues, EtherProv is capable of mitigating unaddressed security threats detected within a deployed contract. Since the execution paths and their static analysis data are known before the contract terminates, EtherProv enables a dynamic modification of the control flow execution for specified execution paths in real-time, e.g., reverting the contract call if a path with specific (e.g., undesirable) properties is encountered. To the best of our knowledge, none of the existing approaches support such mitigation with deployed contracts.

To evaluate EtherProv's capabilities, we show an analysis of three well known security vulnerabilities, i.e., liquid ether, re-entrance, and restricted writes. In comparison with a well-known contract analyzer, Slither [9], EtherProv is capable of detecting all issues *across* contracts. We present a scenario to illustrate our approach's analysis of unaddressed security concerns within already deployed contracts. We further evaluate EtherProv's mitigation capability on the original "King of the Ether Throne" contract. As our evaluation shows, EtherProv can efficiently identify vulnerable contracts with an average contract instrumentation gas overhead of 18.9%.

To summarize, our main contributions are as follows:

- We propose an efficient path profiling approach inspired by Ball et al. [18] to accurately capture the execution flow path across multiple deployed contracts' interaction, with low gas consumption and in real-time. This information is available to the executed contract for real-time security analysis and can be efficiently retrieved using on-chain querying for off-line analysis.
- Equipped with contract execution flow provenance, EtherProv enables to efficiently detect contracts' known security issues, and analyze and mitigate new/unaddressed security issues in deployed contracts. The security issues can span multiple interacting contracts across transaction history. The code is publicly available[1].
- EtherProv uses a unified data schema that enables to utilize static and dynamic data synergy to efficiently analyze new security threats, and extract and incorporate their pattern to be used in detecting future similar known issues.

II. Related work

As the size of Ethereum blockchain grows exponentially, so do the security concerns [19]. Over the past few years a number of tools were developed to verify the security properties of smart contracts. These approaches can be broadly classified into static and dynamic analysis.

a) Static analysis: A number of the security analysis approaches rely on static analysis of the code, i.e., analysis of code without its execution (e.g., Slither [9], Vandal [12]). Static approaches such as Oyente [20], Securify [11], and Mythril [21] utilize symbolic execution for detecting vulnerabilities, i.e., mathematical analysis of different program paths, which is known to suffer from a path explosion problem. On the other hand, Solc-Verify [10] leverages formal verification approach. Using contract code annotated with specifications, Solc-Verify confirms contract's properties using SMT solvers.

b) Dynamic analysis: The dynamic analysis based approaches utilize runtime information for vulnerability analysis. These approaches generally leverage instrumentation to collect run-time metrics such as execution time, instruction count, and gas consumption. Compared to static analysis, dynamic analysis has been less widely adopted for security analysis of smart contracts. These include SODA [14], ECFChecker [22], Sereum [23], HORUS [16] and TXSpector [24].

The existing research studies extract and analyze a contract's execution flow using extensive instrumentation of an Ethereum client [14], [15], [25]. For example, SODA, which is complementary to EtherProv, collects fine grained information at

[1]https://github.com/shomzy/EtherProv

the cost of significant resource consumption. While EtherProv enables collection and analysis of the same data, including the executed path with considerably less instrumentation and without modification of the client node.

The instrumentation approach was also adopted by EVM* [15]. EVM* enables monitoring a contract function's execution in real-time by instrumenting the EVM client node with monitoring and interrupting strategies. In essence, EVM* collects relevant opcodes, and determines if a transaction should be terminated or allowed to execute. EVM* is one of the few systems offering a mitigation strategy for already deployed contracts. However, its reliance on per opcode evaluation is resource consuming and limited to vulnerabilities and bugs contained within a single function. EtherProv, on the other hand, enables an on-line reinforcement by instrumenting the contract's bytecode, without requiring EVM client node modification.

In studies that do not leverage instrumentation, a specific capability of the Ethereum Geth client, which allows to replay any transaction is required (e.g., [16]). In these cases, all relevant past transactions need to be executed to reconstruct the transaction's storage states, which is resource demanding.

c) Provenance: Data provenance refers to the origin and change history of data. In database systems, three primary categories of data provenance [26] have been proposed: *Why-provenance*, *How-provenance* and *Where-provenance*. They refer to the ways in which input tuples are related to the output tuples in a query result. While data provenance deals with data content, workflow provenance is about the transformation process of data that is modeled by a predefined dataflow or control flow. For blockchain, the states of a blockchain can be referred to as data provenance and smart contracts' execution flow can be considered as workflow provenance [27]. EtherProv collects smart contract *execution flow provenance*, which is stored in a provenance database (see Fig. 1) and queried to investigate security issues, among others.

III. The EtherProv system

The EtherProv system is designed (1) to collect provenance information for each function's control flow through static and dynamic analysis; (2) to detect security vulnerabilities, in a single contract and across interacting contracts, by checking compliance with the known security properties; and (3) to provide tracking and mitigation functionality for transactions exploiting vulnerabilities in already deployed contracts. The overview of the system is presented in Fig. 1.

a) Provenance collection: Given a smart contract's source code, EtherProv extracts information on control structures, storage manipulations, and function calls including their corresponding parameters, following each function's CFG. The individual CFGs are then combined to construct an *extended CFG* encompassing the entire contract's control flow spanning multiple functions' interactions across contracts. The extended CFG is then used to encode individual paths and calculate an efficient paths profiling. The encoded paths, in addition to the CFG edges that comprise each encoded path, are stored in the provenance database. To facilitate tracking capabilities and collection of provenance data during contract's execution, the contract's source code is instrumented along with the extended CFG's path. The instrumented contract is then compiled into EVM bytecode and deployed to the blockchain. Upon the contract's execution, the execution's encoded path is emitted, which is collected off-chain, along with additional dynamic data, and stored in the provenance database.

b) Security analysis: The information collected both statically and dynamically enables contracts' security analysis. More specifically, we check a contract's compliance with the security properties of known vulnerabilities and provide context information on the exploitability of this vulnerability along the executed contract flow; across different execution flows of the same contract; and across different execution flows of different contracts across transactions. In this analysis, we leverage the Datalog language syntax.

c) Tracking & Mitigation: The mitigation component is designed to analyze a transaction in real-time based on the collected provenance information. If a transaction execution violates security properties, EtherProv is capable of dynamically reverting control flow execution for specified execution path in real-time, effectively interrupting transaction execution.

A. Smart contract execution provenance collection

EtherProv collects fine-grained smart contract execution provenance through static and dynamic analysis of the Ethereum smart contracts (Fig. 2).

1) Generating extended CFG: Given the contracts' Solidity code, the source code static analyzer extracts static analysis data (e.g., control structures, storage manipulations, function calls, etc) from contracts, derived contracts, contract's functions, variables, interfaces, libraries, following each function's control flow. CFG is represented as a graph of nodes. Each node is provided in two forms of granularity: Static Single Assignment (SSA) form and non-SSA form. Each Solidity source code statement may be represented by a single CFG node in the non-SSA form. In contrast, the SSA form for the same statement may contain multiple SSA nodes. For example, for the single-line Solidity statement `uint variable = array[index];`, the non-SSA form representing the statement consists of a single non-SSA node. The SSA form comprises of at least two SSA nodes: one SSA node for storing the de-referenced array cell in the provided index in a temporary variable and a second SSA node for storing the temporary variable in the lvalue variable. EtherProv stores, for each contract, its Solidity CFG in both SSA and non-SSA forms. The finer granularity of the SSA form is used to analyze all commands constituting a statement. The granularity of the non-SSA form includes the Solidity statements' code and location in the source code, which is used to help instrument the Solidity source code. The mapping between SSA and non-SSA forms are also stored and used to provide static/dynamic analysis capabilities in either granularity. Individual function's CFGs include information on calls to functions inside the same contract or to functions in external contracts, when applicable,

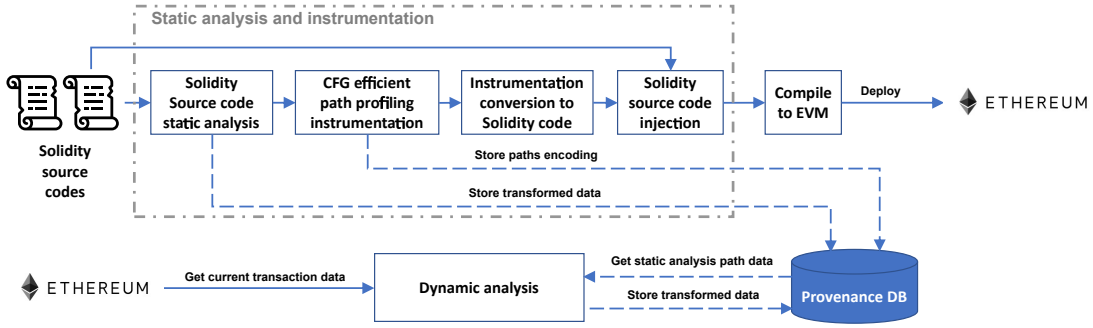
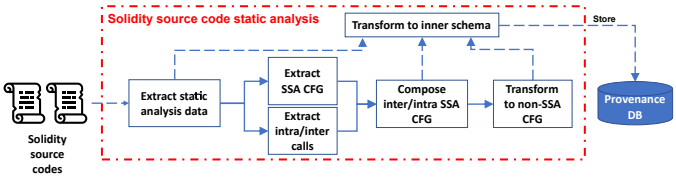
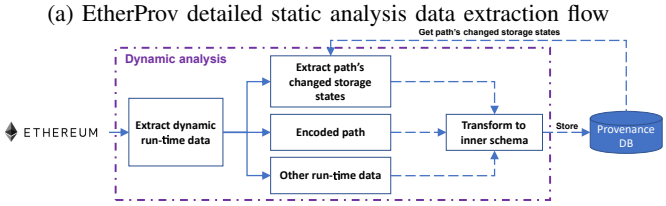


Fig. 1: EtherProv system overview



(a) EtherProv detailed static analysis data extraction flow



(b) EtherProv detailed dynamic analysis data extraction flow

Fig. 2: EtherProv detailed static and dynamic analysis data extraction flow

including constructor calls when inner contract instantiation occurs. In order to enable static analysis of an entire program's CFG spanning multiple functions' interaction across contracts, EtherProv extends each contract function's CFG to encompass intra-/inter-contract calls. This is achieved by adding new edges for each SSA node if it is a call site. One edge is added from the call site SSA node to the first SSA node of the called function. Additional edges are added from each terminating SSA node, e.g., a return statement, in the called function back to the calling call site SSA node. Similarly, the non-SSA CFG is extended with the intra-/inter-contract function call edges. The resulting CFG covering all functions of a contract is referred to as an *extended* CFG. For the rest of the paper, we will refer to an *extended* CFG as a CFG.

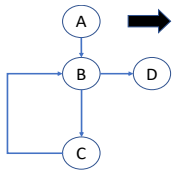
2) CFG efficient path profiling: In order to enable efficient tracing of contracts' execution paths at run-time, EtherProv instruments the (extended) CFG using a path profiling approach inspired by Ball et al. [18]. The need for efficient path profiling stems from the significant overhead typically incurred by instrumentation, which can be unacceptable in the Ethereum environment where contract execution consumes gas of a limited quantity. Ball's algorithm gives an efficient approach to accurately determine the frequency of a control flow path. The algorithm operates on a directed acyclic graph (DAG) with a single main entry node and a main exit node. A path is tracked in the DAG by updating a register along the path execution through instrumentation in selected edges. To achieve this, the algorithm enumerates and uniquely encodes each possible path. As a specific path is executed, the instrumented edges accumulate the path encoding in a register. The traversal of the maximum spanning tree in a reverse direction is manifested with accumulation of negative numbers.

To apply this algorithm in the context of smart contracts, several deficiencies need to be addressed.

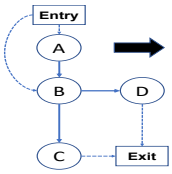
First, a path in our context may span multiple contract interactions. To address this, EtherProv uses a `PathAccumulator` contract containing a single `uint256` storage variable, which serves as the "global" *path accumulator* register (*accumulator* for short) and functions to manipulate it.

Second, each contract instruction consumes some amount of *gas*, a measurement unit used to calculate the amount of funds to be paid for the operation. Ball's algorithm requires to produce a path encoding per iteration, which is expensive in Ethereum context. In order to reduce gas consumption, EtherProv emits the path using the low gas consuming `emit` instruction, which writes the accumulated value to a transaction's event. When a path is emitted, the accumulator is reset to 0, which refunds gas. Further, EtherProv reduces multiple log writes in a loop by compressing repetitive paths encoding.

Third, a Solidity contract may have multiple entry point functions. These functions' identifiers are either *external*, which can only be called from outside the contract, or *public* functions that are not called from any contract. A contract may also have multiple exit points in the form of statements that terminate an entry function (e.g., return statements, loop breaks, uncaught throw statements). Since Ball's algorithm is designed to work on a DAG, we convert the CFG to a DAG by connecting the single DAG's entry node to the CFG's nodes that represent the first statement of each entry function. The DAG's exit node is connected to each CFG node that represents an entry function's terminating statement. Further, each back-edge (e.g., resulting from while loops, recursive function calls) is replaced by two additional edges: one edge pointing from the DAG's entry node to the back-edge's destination node and one edge pointing from the back-edge's source to the DAG's exit node. Fig. 3a shows an example CFG and its corresponding DAG in Fig. 3b. The corresponding encoding of each DAG path is shown in Fig. 3c. Each path, which may be comprised of multiple edges, is encoded as a single value, resulting in a path compression.



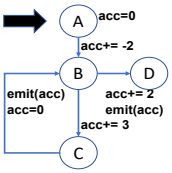
(a) An example of a CFG



(b) The resulting DAG

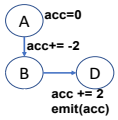
Path	Encoding
A,B,D	0
A,B,C	1
B,D	2
B,C	3

(c) DAG paths encoding

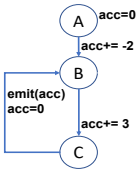


(d) Instrumented CFG

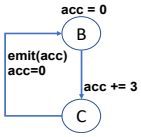
Fig. 3: Efficient path profiling - CFG instrumentation and paths extraction



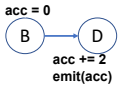
(a) Type I path



(b) Type II path



(c) Type III path



(d) Type IV path

Fig. 4: Efficient path profiling - illustration of four path types

In order to calculate a path encoding during run-time, the generated DAG is instrumented with an accumulator. Each instrumented edge adds a positive or negative value along the taken execution path. At the end of each path execution, the accumulator contains the path's encoding emitted (logged) in the order it was encountered, which correlates to the order of the contract's execution flow. The instrumented DAG edges are then copied to the corresponding CFG edges to form an *instrumented CFG* (see Fig. 3d).

Fourth, an instrumented CFG may contain loops. Paths without loops are encoded as a single value. Paths containing at least one loop are encoded using multiple values. A back edge in the instrumented CFG causes the current accumulated path encoding to be emitted, and the accumulator to be reset, in order to accumulate a new path encoding from that node. There are 4 path types: **I -** a path not containing a loop, **II -** a path starting at the beginning of the entry node and leading to the first loop, **III -** a path after a loop leading to another loop, and **IV -** a path after the last loop leading to the exit node. Fig. 4 shows the 4 path types, which are extracted from the instrumented CFG example shown in Fig. 3d.

Fig. 4a shows a type I path not containing a loop. The accumulator is incremented from 0 to -2 and back to 0 at the path's end, where it is emitted before node D, as the path encoding 0. Fig. 4b shows a type II path entering a loop for the first time. The accumulator is incremented from 0 to -2 to 1 at node C $(-2+3)$, where it is emitted before node B as path encoding 1. The accumulator is then re-initialized to 0 for the accumulation of the next path's part, which can be another loop iteration, leading to a different loop, or leading to the exit node. Fig. 4c shows a type III path, which iterates through a loop. The accumulator is incremented from 0 (after it was re-initialized in the type II path) to 3 at node C $(0+3)$, and is emitted before node B, as path encoding 3. The accumulator is then re-initialized to 0 for the accumulation of the next path. Fig. 4d shows a type IV path, which leads from the last loop iteration to the exit node. The accumulator is incremented from 0 to 2 at node D $(0+2)$, and is emitted as path encoding 2.

When no loop is executed, a single path encoding is emitted. When a loop is executed, a path encoding is emitted for each iteration. To efficiently emit a loop iteration encoding (type III path) at run-time, EtherProv counts the number of loop iterations that were encountered and emits a single type III path encoding along with its count. For example, for a path not containing loops such as A-B-D, the emitted encoding is 0. For a path containing loops such as A-B-C-B-C-B-C-B-D, the encoding is 1 (for A-B-C), 3 (for B-C), 3 (for B-C), 2 (for B-D). The 2 loop iterations (B-C-B-C) are accumulated and emitted as the encoding for B-C along its count. The final compressed path will be emitted as (1), (3:2), (2). The mapping of each encoded path to its instrumented CFG related edges is stored in the provenance database as a Datalog fact.

3) Instrumenting Solidity source code: The instrumented CFG provides efficient and accurate profiling of CFG paths. To facilitate provenance collection during the contract's execution, we translate each of the CFG's edge instrumentations to their corresponding Solidity statements.

Since each CFG node represents a Solidity statement with a known location, EtherProv heuristically determines whether instrumentation should be injected after the source statement or before the destination statement. When mandated by the instrumentation, additional code is injected. For example, an *if* statement without an *else* clause, may have an instrumented edge corresponding to the *false* branch, in this case an *else* clause is injected with the instrumented code.

Additional dynamic execution data is extracted with further Solidity code instrumentation, which includes a getter function for each of the contract's public/private storage state variables. The modified Solidity source code is then compiled to EVM bytecode and deployed to the Ethereum blockchain.

The accumulator that holds path encoding is instrumented through the use of a `PathAccumulator` contract, that is created beforehand. Its address is injected to each contract's constructor in the original Solidity source code, in addition to code that saves the PathAccumulator contract in an internal contract storage state.

4) Dynamic data extraction: EtherProv enables the extraction of an execution flow, consisting of high-level Solidity code

statements by instrumenting only the contract's bytecode. The contract's Solidity code is instrumented to emit an encoded execution flow path of a contract in real-time to the transaction log. The encoded path can be then extracted from any historic transaction containing an instrumented contract. The contract's exact execution flow is decoded from the encoded path by querying the provenance database path mapping. The contract's executed path encoding is extracted from the blockchain by querying the PathAccumulator contract's events, which are located in the executed transaction.

To enable more advanced analysis capabilities, EtherProv additionally collects different types of dynamic provenance data, i.e., the call function, its parameter names and types, and the contract's name are extracted from the contract Application Binary Interface (ABI); block and transaction numbers are extracted from the mined block; from address, to address, called function name and parameter values are extracted from the mined block's transaction.

To enable an analysis, which requires a contract's current storage state values and current Ether balance, EtherProv is run after each transaction to collect the current values, which may change in the subsequent transactions. The current contract storage variable values are extracted from the contract's instrumented getters, and the current Ether balances are extracted from the deployed contract. To save computation and space, EtherProv queries the static analysis data from the data provenance database to: (1) decode the fully executed path from the extracted encoding, and (2) to extrapolate only the storage states that were read from or written into, according to the decoded executed path.

B. Security analysis

To enable security analysis and debugging, Datalog rules are issued against EtherProv's provenance database. The provenance database is itself comprised of an extensional database (fact tables) and intensional database (derived by rules).

The fact tables are comprised of static and dynamic analysis related tables that are populated in the corresponding phases. Static analysis related tables include information on contracts, their functions and parameters, CFG nodes in SSA/non-SSA form, variables throughout contracts and their functions, paths and their related state access details, etc. Dynamic analysis related tables include information on contract to address mapping, contract call and its parameters, extracted path and its related read/written state parameters.

The intensional database comprises the security and debugging analysis rules, which use the extensional database (table facts). Section IV contains examples of rules enabling the security and debugging analysis.

C. Tracking & Mitigation

Since a deployed Ethereum contract is immutable, addressing unhandled security issues or bugs is challenging. EtherProv tracks the executed contract's path in real-time analyzing its executed path, hence, knowing its outcome before the transaction commits. The collected provenance data, e.g., current

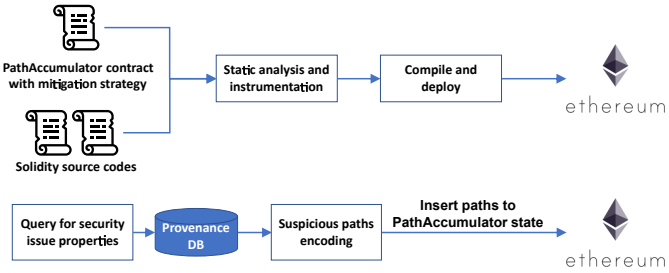


Fig. 5: EtherProv tracking & mitigation overview

executed path's input and output storage state parameters, local and global variables, function parameters, etc., are used to identify and collect auxiliary data to help inform a mitigation action in real-time. Such mitigation can modify the current execution flow, e.g., revert the current transaction or call additional internal/external functions, if the transaction's path is determined to be suspicious or malicious.

The related paths corresponding to the known properties of a security or debugging issue can be retrieved from the provenance database, and consequently stored in the blockchain's state, such as a dictionary/mapping, to be queried by the mitigation component in real-time. Fig. 5 shows the tracking and mitigation deployment steps.

IV. Security Evaluation

A. Implementation

EtherProv was implemented using Python language with Ganache [28] as an Ethereum blockchain for the deployment and execution of smart contracts. Slither [9] was used as a third-party Solidity source code static analysis tool and Souffle [29] as the Datalog query engine to run user-defined analysis queries against the EtherProv's provenance database.

B. Detecting known smart contract vulnerabilities

Many vulnerabilities were discovered in smart contracts. In this work, we review 3 of them and show how EtherProv's capabilities help to reason about them.

1) Liquid ether: Ethereum contracts have a capability of sending and receiving Ether. The bugs related to this category permanently lock funds. One of the vulnerabilities that fall in this category is Parity Wallet bug [30]. In 2017, the removal of a library from the Ethereum blockchain, which was used to exclusively send Ether to other contracts, by a referencing contract, caused an entrapment of $160M worth of Ether [30].

EtherProv verifies this security issue does not occur by checking if a contract can send Ether by either (1) suicide/self destruct function, or (2) a call function with a positive number. Listing 1 shows EtherProv's corresponding Datalog rules.

```
1 liquid_ether_compliance(node_id) :-
      node_with_liquefiable_function_calls(node_id).
2 liquid_ether_compliance(node_id) :-
      node_with_call_value_not_0(node_id).
3 liquid_ether_compliance(node_id) :-
      node_with_call_value_dependent_on_sender(node_id).
4
5 node_with_liquefiable_function_calls(node_id) :-
6   non_contract_function_call(node_id, "suicide(address)").
7 node_with_liquefiable_function_calls(node_id) :-
```

```
  non_contract_function_call(node_id,
    "selfdestruct(address)").

node_with_call_value_not_0(node_id) :-
  node(node_id,call_value),
  node_variable(call_value,name,"uint256","True"),
  name != "0".

node_with_call_value_dependent_on_sender(node_id) :-
  node(node_id,call_value),
  node_variable(call_value,_,"uint256","False"),
  variable_may_depend_on(call_value,"Client#msg.value").
```

Listing 1: Verifying liquid ether compliance (EtherProv Datalog rules)

2) Re-entrancy: While re-entrancy can occur in many forms, it typically requires a contract to 'call' another contract or external function multiple times before its previous invocations were finalized, i.e., "re-enter" a function. Re-entrancy vulnerabilities can occur across multiple functions and multiple contracts. This can cause severe damages, including fully draining funds from vulnerable contracts.

An infamous example of re-entrancy bug happened in 2016, the vulnerable DAO contract was exploited, resulting in the stealing of $60M worth of Ether [31]. The contract included a function call, which sent Ether to the recipient. The function determined the amount to be sent by inspecting a storage variable, which was updated with the remaining amount information after the function call. The attacker re-invoked the function multiple times, i.e., sending ether, before its dependent storage variable could be updated.

EtherProv verifies this security issue does not occur by checking that there is no write to a storage value after *any* call instruction. Listing 2 shows EtherProv's Datalog rules. Rows 1-5 contain the main rule `no_writes_after_calls_violation`. It is true if the contract (1) contains a call-function, and (2) if any of its following instructions are writes to a storage variable.

`no_writes_after_calls_violation` is true when the node *`node_id`* is a call-function (identified in the schema as `"LowLevelCall"`), and is followed by a node *`followed_by_node_id`* that contains an lvalue storage variable (*`lvalue_node_variable_id`* in `nodes_with_storage_writes`).

```
no_writes_after_calls_violation(node_id,
    followed_by_node_id, lvalue_node_variable_id) :-
  node(node_id,"LowLevelCall"),
  node_may_be_followed_by(node_id,followed_by_node_id),
  nodes_with_storage_writes(followed_by_node_id,
      lvalue_node_variable_id).

nodes_with_storage_writes(node_id,
    lvalue_node_variable_id) :-
  node_operation_with_l_value(node_id,
      lvalue_node_variable_id),
  node(node_id,"Assignment"),
  node_variable(lvalue_node_variable_id,"NULL","True").
nodes_with_storage_writes(node_id,
    lvalue_node_variable_id) :-
  node_operation_with_l_value(node_id,
      lvalue_node_variable_id),
  node(node_id,"Assignment"),
  node_variable(lvalue_node_variable_id,points_to,_),
  points_to != "NULL",
  node_variable(points_to,"NULL","True").
```

Listing 2: Verifying no writes after calls compliance (EtherProv Datalog rules)

3) Restricted writes: Ethereum contracts can be accessed publicly, hence ensuring that only authorised users can modify the contract is essential. Failing to ensure this can enable unrestricted users to change the contract's behaviour and, in some cases, steal its Ether. Such vulnerability was exploited in 2017 when an attacker was able to update the contract's owner to his own address, resulting in a theft of $30M [32].

EtherProv detects this security issue by checking if a contract either (1) contains a path, which is not a result of a branch, that enables to change a state by any user (independent of sender), or (2) contains at least one path that is a result of a branch, where the branching condition is independent of the sender and leads to a storage write. Listing 3 shows EtherProv's Datalog rules.

```
restricted_writes_violation(batch_id, root_node_id) :-
  root_path_yes_branches_with_write_independent_of_sender(
      batch_id, root_node_id, _, _).

restricted_writes_violation(batch_id, root_node_id) :-
  !root_of_path_with_branches_dependent_of_sender(batch_id,
       root_node_id, _, _),
  root_path_yes_branches_with_write_independent_of_sender(
      batch_id, root_node_id, _).
```

Listing 3: restricted_writes_violation (EtherProv Datalog rules)

4) Detecting violations across contracts: The existing static analysis tools can analyze only single contracts. Yet, some security issues span over multiple interacting contracts. Listing 4 provides an example of a *No writes after calls* vulnerability, which spans multiple contracts. The Client contract (lines 1-13) uses the LockManager contract (lines 14-19). The Client's function `transferFundsOnce` receives an address to send 100 Ether to. If the lock is set to `false` (line 5), the code continues to send 100 Ether to the `dest_address` address (line 8) and sets the lock to `true` (line 10). Since the lock remains locked, future calls to the function do not send Ether. An attacker can drain Ether from this contract if he is the first caller to the `transferFundsOnce` function and the provided `dest_address` is an address of a contract with a payable fallback function, which contains a call to the `transferFundsOnce` function with its own address. When the function is first invoked, the lock's state is `false`, and the Ether is sent to the attacker's contract (line 10), where the fallback function is invoked, which in turn calls the `transferFundsOnce`. Since the lock's state is not changed yet, an additional 100 Ether are sent to the attacker's contract with the repeated invocation of the fallback function.

We executed the code shown in Listing 4 in Slither [9]. Slither was not able to detect this vulnerability, while EtherProv's analysis that spans multiple contracts, was able to detect it.

```
contract Client {
```

```
  LockManager _lm = LockManager();
  uint _balance = 1000000;
  function transferFundsOnce(address dest_address) {
    if (!_lm.isLocked()) {
      if (_balance > 100) {
        _balance = _balance - 100;
        dest_address.call.value(100)();
      }
      _lm.lock();
    }
  }
}
contract LockManager {
  bool _locked = false;
  constructor() public {}
  function isLocked() returns (bool) {return _locked;}
  function lock() {_lock = true;}
}
```

Listing 4: Verifying no writes after calls across multiple contracts compliance

C. Analyzing new security threats in deployed contracts

Scenario: Bank's operations are managed by the Bank contract, while each user's bank account's operations are managed by the Client contract. Both are created and deployed to the blockchain. The user's Client contract is used to send/receive funds to/from other Client contracts.

When a user wants to send funds to a designated Client contract, he issues the request to his own Client contract with the amount and address of the designated Client contract's address. The user's Client contract queries the Bank contract for the fee, sends it to the Bank contract, and sends the funds to the designated Client contract. This process is implemented in the Client contract's `_sendAmount` function (Listing 5).

```
function _sendAmount(uint amount, Client toClient) {
  uint fee = _bank.getCurrentFeeContractAPI(_balance,
      _accountType);
  uint newBalance = _balance - amount - fee;
  if (newBalance >= 0) {
    toClient.addAmountContractAPI(amount);
    _bank.depositFeeContractAPI(fee);
    _balance = newBalance;
  }
}
```

Listing 5: Solidity Client contract's internal _sendAmount function

Security concern 1: The bank has created 2 Client contracts: client1 and client2 for user1 and user2, respectively. Both users should have a "preferred" account type. User1 deposits 2000 credits into his Client1 contract, and then issues a request to its Client1 contract to transfer 100 funds to the Client2 contract. As a result, User1 sees that the fee deducted from his Client contract's balance is larger than it should be.

Security concern 1 analysis: EtherProv provides sufficient provenance data to dynamically examine data flow across these deployed contracts. The full analysis encompasses several queries, for brevity we only focus on verification of balance before and after the contract call. Listing 6 provides the EtherProv's query that extracts the value of all state changes resulting from contract call (`"9f0efee0..."`), which include $current\_value = 1884$ and $prev\_value = 2000$. From this the fee can be calculated as $2000 - 1884 - 100 = 16$, which is different from the Bank contract's fee related to the "preferred" account.

row	path_id	edge_id	from_node_id	to_node_id	type	expression
1	6	252	Client#sendAmount...	Client#sendAmount	EXPR	sendAmount(amount,toClientAddress)
2	6	231	Client#sendAmount	Client#_sendAmount	EXPR	_sendAmount(amount,toClient)
3	6	236	Client#_sendAmount	Bank#getCurrentFee...	NEW	fee = _bank.getCurrentFee...
4	6	199	Bank#getCurrentFee...	Bank#getCurrentFee...	NEW	sendFee = 0
5	6	200	Bank#getCurrentFee...	Bank#getCurrentFee...	NEW	underMinBalanceFee = 0
6	6	201	Bank#getCurrentFee...	Bank#getCurrentFee...	NEW	isPreferred = keccak256(bytes)(ab...
7	6	202	Bank#getCurrentFee...	Bank#getCurrentFee...	IF	isPreferred
8	6	203	Bank#getCurrentFee...	Bank#getCurrentFee...	EXPR	sendFee = _premiumAccSendFee
9	6	204	Bank#getCurrentFee...	Bank#getCurrentFee...	IF	balance < _premiumAccMinBalance
10	6	205	Bank#getCurrentFee...	Bank#getCurrentFee...	EXPR	underMinBalanceFee = _premiumAccUnd...

Fig. 6: Sampled results of decoded path from EtherProv query

```
contract_call_changed_states(state_id, current_value,
    prev_value) :-
  contract_call_written_states(_, "9f0efee0...", state_id,
      current_value, prev_written_state_id),
  contract_call_written_states(prev_written_state_id, _, _,
      prev_value, _).
```

Listing 6: Contract call changed states (EtherProv query)

To understand the difference, the contract execution flow provenance of the contract call involving the deployed Client and Bank contracts should be analyzed (Listing 7).

```
decoded_path(path_id, edge_id, from_node_id, to_node_id,
type, expression) :-
  dynamic_path("9f0efee0...", path_id, _, _ ),
  static_path(path_id, edge_id),
  static_edge(edge_id, from_node_id, to_node_id),
  static_node(from_node_id, _, type, expression).
```

Listing 7: Decoded path (EtherProv query)

A sample of the query results are shown in Fig. 6. Rows 7-8 show that the branch regarding the "premium" account was taken instead of the branch regarding the "preferred" account. The analyst continues to query the Client's accountType state value and discovers it was entered with a typo as the "prefered" account type.

Security concern 2: The Client contract contains a bug in `_sendAmount` (Listing 5 row 3). The statement `uint newBalance = _balance - amount - fee;` contains an assignment to a variable of type `uint` . A negative number assignment to a `uint` results in an integer underflow, which consequently may result in a very large number. For example, a user sends funds to a Client where the sum of the fee and the amount are larger than the balance. Following the transaction, the user will obtain a large balance amount. Consequently, all transactions that involve this large balance will contribute to the magnitude of the issue.

Security concern 2 analysis: This integer underflow becomes obvious when querying specific paths, e.g., paths that contain a call to `_sendAmount` with focus on states of type `uint` that are known to be strictly decreasing (e.g., `balance` variable when sending funds). If the states' values are increasing between subsequent transactions, an integer underflow is detected.

EtherProv enables to detect tainted data by following the flow of tainted values, i.e., when a contract call operates on tainted values, its output storage states are also considered tainted. In the integer underflow example, the first occurrence of the (tainted) large balance value can be used to query all

Contracts count	Avg gas overhead	Avg gas overhead std
1652	18.90%	0.378

TABLE I: Instrumented contracts statistics

contract calls that used either this value directly or other values tainted by this value (Listing 8 where `"8093c811..."` is contract state id).

```
1 tainted_state_ids(batch_id, tainted_state_id) :-
2 tainted_state_ids("8093c811...").
3
4 tainted_state_ids(new_tainted_state_id) :-
5   tainted_state_ids(known_tainted_state_id),
6   state_parameter_read(contract_call_id, _,
        known_tainted_state_id),
7   state_parameter_written(_, contract_call_id, _, _,
        new_tainted_state_id).
```

Listing 8: Extracting tainted state ids (EtherProv query)

The recursive query retrieves all contract calls that read a tainted contract state. For each such contract call, the written state values are also considered tainted and are added to the "known tainted states". The query continues recursively until no new tainted states are added.

D. Mitigating security threats in deployed contracts

Fixing an unhandled security issue or bug in a deployed contract is challenging, as a deployed contract is immutable. EtherProv enables mitigation of such issues. Listing 9 shows an example of handling an issue by reverting the specified paths. The Listing shows partial `PathAccumulator` code related to emitting the accumulated executed encoded path in run-time (called from instrumentation). `PathAccumulator` uses a `_revert_paths` dictionary to store all paths that should be reverted in run-time before completing the transaction. Lines 6-8 show the function used by the `PathAccumulator` owner to update the dictionary with those predefined paths. Upon transaction execution, before emitting the current executed path, if `_revert_paths` contains the path (line 2), the transaction is reverted.

```
1 function flush_path_data() {
2   if (_revert_paths[_current_path.path_id] != 0) revert();
3   emit path(_current_path.path_id, _current_path.count);
4   _current_path.is_init = 0;
5 }
6 function update_reverted_path_id(uint[] calldata path_ids,
      uint size) {
7   for (uint i=0; i<size; i++) _revert_paths[path_ids[i]]=1;
8 }
```

Listing 9: PathAccumulator partial Solidity code

We evaluated EtherProv's mitigation capability on the original "King of the Ether Throne" contract, KingOfTheEtherThrone.sol [33]. The contract was instrumented and deployed. We then issued a query that extracted all paths encoding containing a call to the `claimThrone` function, which were then inserted to the `_revert_paths` dictionary. The subsequent contract calls were reverted successfully.

V. Performance Evaluation

In this section we evaluate the instrumented contracts' execution gas overhead. The contracts' Solidity source code

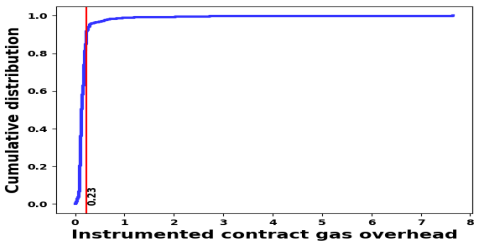


Fig. 7: Instrumented contracts' average overhead CDF

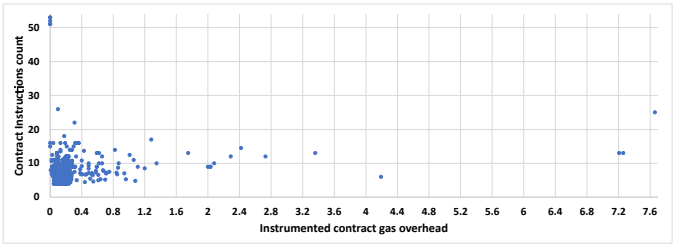


Fig. 8: Instrumented contracts

were retrieved from etherscan.io [34] and the evaluation was performed on the Ganache platform. For each contract, we identify its external functions and public functions that are not called by any contract. Each such function is executed with its instrumented and uninstrumented versions for which we calculate the respective gas overhead. When executing an instrumented contract's function we extract from the emitted executed path the number of executed instructions, which is the number of executed instructions in the uninstrumented contract. We calculate a contract's execution gas overhead as $\frac{\sum_{f \in Contract}(f^{gas}_{instrumented} - f^{gas}_{uninstrumented})}{\sum_{f \in Contract}(f^{gas}_{uninstrumented})}$, where f is a contract's executed function. The contract's corresponding number of instructions ($instructions\_count$) are calculated as the average of all executed uninsturmented contract functions' instructions number. As the results in Table I show, the average contract gas overhead is 18.9%. In comparison, the study by Wang et al. [35] reported an average of 28.27% run-time overhead. Fig. 7 shows that 90% of the contracts (vertical red line) incur a maximum gas overhead of 23%.

The instrumentation of a contract containing loops uses a path compression logic before emitting a loop encoding. Our instrumentation of contracts without loops is lightweight and hence incurs considerably lower gas cost. Fig. 8 helps to better understand the varying instrumented gas overheads. We manually analyze the three corner cases in the graph:

- *High instructions count and low instrumented contract gas overhead* (top left of the graph). We analyzed the top 5 contracts with the highest ratio of $\frac{instructions\_count}{instrumented\_contracts\_gas\_overhead}$. Most of the executed functions' paths in the extracted uninstrumented contracts contained a high number of uninstrumented instructions alongside low instrumentation due to lack of branching or loops, which helps explain the low gas overhead. Specifically, some functions' paths contained 50-60 instructions (e.g., write hard coded lists into a map storage). Since storage write is the most expensive instruction anyways, its contribution to the

instrumentation gas overhead is negligible.

- *Low instructions count and low instrumented contract gas overhead* (bottom left of the graph). We examined the top 5 contracts with the lowest instrumented contracts gas overhead and the lowest instruction count. Most of the executed functions' paths in the extracted uninstrumented contracts contained relatively few instructions and low instrumentation (due to lack of branching or loops), which helps explain the low instrumentation gas overhead. Specifically, some function paths contained 5-13 write to storage instructions (most gas consuming), which helps explain the considerably lower gas overhead.
- *Lower instructions count and high instrumented contract gas overhead* (bottom right of the graph). We analyzed the top 5 contracts with the highest ratio of $\frac{instrumented\_contracts\_gas\_overhead}{instructions\_count}$. Most of the executed functions' paths contained relatively few instructions and high instrumentation due to loops and branching, which helps explain the high instrumentation gas overhead. Specifically, some executed function paths contained 1-2 storage or memory reads/writes or an emit instruction (all of which consume little gas) and 1-2 branches and a loop, which contribute to higher instrumentation and thus explain the considerably higher instrumentation gas overhead.

We examined some of the instrumented contracts in between the corner cases and found that the ratio of instrumentation to the cost and frequency of uninstrumented instructions correlates with the logic discussed in the corner cases, i.e, more/less costly executed instructions with corresponding less/more branching or loops result in low/high gas overhead respectively. The results show that instrumentation is the most efficient on contracts without loops. However, contracts with loops incur a gas overhead relative to the loops' frequency.

VI. Conclusion

The rising adoption of blockchain technologies has resulted in companies employing blockchain to manage various valuable assets. Many blockchain platforms support automated execution of smart contract code, and it is very important that smart contracts are free from security vulnerabilities. With the proliferation of such smart contracts, the number of incidents related to smart contract vulnerabilities is also increasing. Existing approaches to analyze smart contract security properties are usually resource consuming and are not suitable for practical deployment. Also, these approaches offer no mitigation strategies with already deployed contracts. To address the limitations of existing approaches, we presented EtherProv, which tracks smart contract execution flow provenance by leveraging both static and dynamic analysis of Solidity source code. Our system enables the tracing of an execution flow spanning multiple interacting deployed contracts with an average instrumentation overhead of 18.9%. We also developed an efficient path profiling approach for the extraction of a contract's executed path. Moreover, EtherProv is capable of mitigating unaddressed security threats detected within already deployed contracts. Our experimental evaluation demonstrates that EtherProv is able to accurately detect several security vulnerabilities, including liquid ether, re-entrancy and restricted writes, and analyze transaction-based security threats in deployed contracts.

References

[1] "Gartner blockchain-platforms. https://www.gartner.com/reviews/market/blockchain-platforms."
[2] "ethereum.org. https://ethereum.org/."
[3] L. Goodman, "Tezos: A self-amending crypto-ledger position paper," 2014.
[4] "eos.io. https://eos.io/."
[5] "cardano. https://cardano.org/."
[6] "Hyperledger fabric. https://github.com/hyperledger/fabric."
[7] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "SmartCheck: Static Analysis of Ethereum Smart Contracts," in *WETSEB*, 2018, pp. 9–16.
[8] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "ZEUS: analyzing safety of smart contracts," in *NDSS*, 2018.
[9] J. Feist, G. Greico, and A. Groce, "Slither: A Static Analysis Framework for Smart Contracts." IEEE Press, 2019.
[10] Á. Hajdu and D. Jovanović, "Solc-verify: A Modular Verifier for Solidity Smart Contracts," in *VSTTE*, 2020.
[11] P. Tsankov *et al.*, "Securify: Practical Security Analysis of Smart Contracts," in *CCS*, 2018.
[12] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, "Vandal: A scalable security analysis framework for smart contracts," *arXiv preprint arXiv:1809.03981*, 2018.
[13] Y. Ding, C. Wang, Q. Zhong, H. Li, J. Tan, and J. Li, "Function-level dynamic monitoring and analysis system for smart contract," *IEEE Access*, 2020.
[14] T. Chen *et al.*, "SODA: A generic online detection framework for smart contracts," in *NDSS*, 2020.
[15] F. Ma, Y. Fu, M. Ren, M. Wang, Y. Jiang, K. Zhang, H. Li, and X. Shi, "EVM*: from offline detection to online reinforcement for ethereum virtual machine," in *SANER*, 2019, pp. 554–558.
[16] C. F. Torres, A. K. Iannillo, A. Gervais, and R. State, "The Eye of Horus: Spotting and Analyzing Attacks on Ethereum Smart Contracts," *arXiv preprint arXiv:2101.06204*, 2021.
[17] "king of the ether. https://www.kingoftheether.com/postmortem.html."
[18] T. Ball and J. R. Larus, "Efficient path profiling," in *MICRO*, 1996.
[19] H. Chen, M. Pendleton, L. Njilla, and S. Xu, "A survey on ethereum systems security: Vulnerabilities, attacks, and defenses," *ACM Comput. Surv.*, 2020.
[20] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," 2016, p. 254–269.
[21] "Mythril classic," 2021. [Online]. Available: https://github:com/ConsenSys/mythril-classic
[22] S. Grossman *et al.*, "Online detection of effectively callback free objects with applications to smart contracts," *Proc. ACM Program. Lang.*, 2017.
[23] M. Rodler, W. Li, G. Karame, and L. Davi, "Sereum: Protecting existing smart contracts against re-entrancy attacks," *NDSS*, 2019.
[24] M. Zhang, X. Zhang, Y. Zhang, and Z. Lin, "TXSpector: Uncovering attacks in ethereum from transactions," in *USENIX Security*, 2020.
[25] T. Chen *et al.*, "DataEther: Data exploration framework for Ethereum," in *ICDCS*, 2019, pp. 1369–1380.
[26] C. L. Cheney, J. and W. Tan, "Provenance in databases: Why, how, and where," *Foundations and Trends® in Databases*, pp. 379–474, 2009.
[27] S. Linoy, S. Ray, and N. Stakhanova, "Towards eidetic blockchain systems with enhanced provenance," in *ICDEW*, 2020, pp. 7–10.
[28] "Ganache. https://www.trufflesuite.com/ganache."
[29] H. Jordan, B. Scholz, and P. Subotić, "Soufflé: On synthesis of program analyzers," in *CAV*, 2016, pp. 422–430.
[30] "Parity security alert. https://www.parity.io/security-alert-2/," 2017.
[31] "The DAO attacked: Code Issue Leads to 60 Million Ether Theft." 2016.
[32] "An In-Depth Look at the Parity Multisig Bug. https://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/."
[33] "King of the ether throne. https://github.com/kieranelby/kingoftheetherthrone."
[34] "etherscan.io. https://etherscan.io/."
[35] X. Wang, J. He, Z. Xie, G. Zhao, and S.-C. Cheung, "ContractGuard: Defend ethereum smart contracts with embedded intrusion detection," *IEEE TSC*, pp. 314–328, 2019.