Detecting Cross-Function Reentrancy from EVM Traces

Semia Guesmi
Ca' Foscari University of Venice, Italy
University of Camerino, Italy

Carla Piazza
University of Udine, Italy

Andrea Gasparetto
Ca' Foscari University of Venice, Italy

Matteo Rizzo
Ca' Foscari University of Venice, Italy

Sabina Rossi
Ca' Foscari University of Venice, Italy

Abstract

Reentrancy remains one of the most critical vulnerabilities affecting Ethereum smart contracts. While many existing analysis tools focus on detecting classical single-function reentrancy, more complex forms such as cross-function reentrancy are harder to identify because they depend on execution semantics and interactions between multiple functions. In this work, we study reentrancy at the level of Ethereum Virtual Machine (EVM) execution traces. We extend the TxSpector framework with new Datalog-based detection rules designed to capture cross-function reentrancy patterns. To support this analysis, we also modernize the trace extraction component by adapting it to recent versions of the Ethereum client and updated EVM instructions. The proposed approach is evaluated on real Ethereum on-chain transaction traces. The results show that our method is able to detect cross-function reentrancy behaviors that are not captured by the original TxSpector rules, demonstrating the effectiveness of pattern-based logic detection at the EVM execution level.

2012 ACM Subject Classification Security and privacy → Software and application security

Keywords and phrases Blockchain, smart contract, Reentrancy detection, EVM, design Patterns, logic rules

Digital Object Identifier 10.4230/OASIcs.FMBC.2026.8

1 Introduction

Blockchain technology has evolved rapidly since the introduction of Bitcoin in 2009 [25], enabling not only decentralized payments but also the execution of programmable logic through smart contracts. Smart contracts are programs deployed on a blockchain whose execution is enforced by consensus, allowing mutually distrusting parties to interact without trusted intermediaries. While this paradigm offers transparency and autonomy, it also introduces severe security challenges. Smart contracts often hold cryptocurrencies of significant value, which makes them attractive targets for malicious actors. At the same time, once deployed they are immutable and cannot be patched; therefore, vulnerabilities discovered after deployment cannot be eliminated by modifying the vulnerable code. Among smart contract vulnerabilities, reentrancy is one of the most critical. It occurs when a contract transfers control to an external entity before completing its own execution, allowing the callee to re-enter the caller within the same transaction and exploit its intermediate state. The DAO attack in 2016 [37], which resulted in the loss of more than three million Ether,

represents the first and simplest instance of this vulnerability. Since then, research has shown that reentrancy can occur in more complex and compositional forms, involving subtle interactions between control flow and state updates.

In response to reentrancy threats, numerous approaches and tools have been proposed to detect and mitigate such vulnerabilities. Formal analysis techniques commonly rely on methods such as symbolic execution, taint analysis, fuzzing, and constraint solving [29]. Beyond bug-finding, smart-contract security has also been studied through information-flow properties: noninterference-based analyses aim at formally ruling out unwanted influence across security levels in contract behaviors [16, 5]. The notion of interference has been studied in other formal settings as well, including behavioral preorders explicitly sensitive to interference; we build on this general intuition and instantiate it to EVM-level smart-contract executions [8, 13]. Several well-known static analysis tools, including Oyente [22], Securify [39], and Mythril [33], analyze smart contract code to identify patterns associated with reentrancy vulnerabilities. Ethor [32] extends this line of work by abstracting the semantics of EVM bytecode using Horn clauses to verify reachability properties. These approaches are complemented by dynamic analysis frameworks such as ContractFuzzer [17] and sFuzz [26], which explore contract behavior at runtime using fuzzing techniques. More advanced analyzers, such as Clairvoyance [43] and SmartDagger [20], specifically focus on detecting vulnerabilities arising from cross-contract interactions.

Most existing tools focus on detecting single-function reentrancy, where the callee re-enters the same function while the contract is in an inconsistent intermediate state. In this work, we analyze reentrancy at the EVM bytecode level using execution traces. We extend Txspector [28], which currently detects vulnerabilities such as Reentrancy, UncheckedCall, and Suicidal contracts but is limited to single-function reentrancy. Our goal is to extend its detection capabilities to identify a more advanced and dangerous form of attack, cross-function reentrancy, where the callee re-enters a different function before the initial execution completes. This work is guided by the following research question: how does reentrancy manifest at the level of EVM execution semantics, particularly in complex and compositional scenarios?

Based on the observations above, this paper makes the following contributions:

1. We define new patterns for cross-function reentrancy, and based on these patterns, we design a set of logic detection rules for its identification.
2. We modernize the TxSpector framework by updating its source code to support recent libraries and EVM changes introduced by newer Ethereum hard forks.
3. We implement the proposed rules in the TxSpector detector component using Datalog and evaluate them through experiments on real-world Ethereum transactions.

2 Background

Smart contracts in Ethereum execute on the Ethereum Virtual Machine (EVM), a stack-based runtime environment. They are typically written in high-level languages such as Solidity [35] and compiled into bytecode, which is the only representation available after deployment unless source code is explicitly published [4]. The bytecode is distributed via contract creation transactions and executed by every node in the network whenever a contract function is invoked. The EVM is designed to ensure deterministic execution across all nodes and follows a stack-machine architecture, where instructions operate on a data stack or constant operands. During execution, a contract can use three memory regions. The *stack* is a volatile region manipulated by dedicated instructions. *Memory* is a volatile, byte-addressable heap that

persists only during a transaction. *Storage* is a persistent key-value store mapping 256-bit words to 256-bit words and represents the only state preserved across transactions. Each contract invocation occurs within a transaction. Transactions can be issued by external accounts or by other contracts. To prevent abuse and ensure termination, Ethereum employs a gas mechanism, where each instruction consumes a predefined amount of gas paid by the transaction sender. This mechanism also protects the network from infinite loops.

In Ethereum, smart contracts interact with external contracts and transfer Ether through message calls using primitives such as call, transfer, and send. These instructions trigger the fallback (or receive) function of the recipient contract, which may execute additional operations depending on the gas forwarded. The `send` and `transfer` instructions forward a limited gas stipend of 2300 gas, restricting the callee's ability to perform complex operations. In contrast, `call` forwards all remaining gas by default, enabling arbitrary code execution and making it the primary source of reentrancy vulnerabilities. Other call-related instructions also exist such that `delegatecall` executes the code of another contract in the context of the caller, meaning that the caller's storage, address, and balance are preserved, and `Staticcall` performs a read-only external call and does not allow state modification during execution. In Solidity, such interactions are commonly expressed such as `a.call{value: x}("fun")`, where `a` is the callee address, the string specifies the function signature, and `x` is the amount of ether transferred.

Prior work has classified smart contract vulnerabilities into three layers: the blockchain layer, the virtual machine layer, and the solidity layer [31]. In this work, we argue that reentrancy detection fundamentally belongs to the virtual machine layer because reentrancy is not only a coding mistake, but a runtime behavior induced by the EVM call semantics, which temporarily transfers control to external code and can dynamically alter the transaction's control flow before the original execution completes [34]. As a result, reentrancy depends on the concrete sequence of call frames, returns, and storage operations during transaction execution, and is therefore more naturally captured at the bytecode and execution level rather than solely at the source-code level. More generally, our choice to perform bytecode-level analysis is motivated by several additional reasons. First, the source code of deployed contracts is often unavailable, as the blockchain stores only the compiled bytecode. Second, certain properties relevant to vulnerability analysis are observable only at the bytecode level (e.g., gas consumption defined by EVM instructions). Third, compiler optimizations may alter the structure of the generated bytecode, which can affect the reliability of analyses performed at the source-code level, making post-compilation analysis more appropriate.

3 Related Work

Smart contract vulnerabilities have been widely studied, with reentrancy recognized as one of the most severe threats. Prior work classifies vulnerabilities into the blockchain, virtual machine, and Solidity layers [31]. Recent work has also investigated reentrancy from a formal information-flow viewpoint, modeling reentrant behaviors through noninterference to capture semantic dependencies across executions [5]. Reentrancy fundamentally arises at the EVM layer, as it is caused by execution semantics and inter-contract interactions rather than source-level syntax, motivating low-level analysis approaches.

Static analysis tools have been extensively used for reentrancy detection. Vandal [7] analyzes EVM bytecode by translating semantic relations into Datalog, enabling scalable detection of multiple vulnerabilities, including reentrancy. Securify [39] follows a similar logic-based approach but applies conservative violation patterns that often flag any state

update after an external call as vulnerable, leading to false positives. Source-level tools such as SmartCheck [36] and Slither [12] rely on syntactic or rule-based detection, which limits their ability to capture complex execution behaviors and cross-contract interactions.

Symbolic execution has also played a central role in smart contract analysis. Oyente [3], followed by Mythril [9], Manticore [38] and Osiris [19], explores execution paths at the EVM level to detect reentrancy and other vulnerabilities. However, symbolic execution suffers from path explosion and scalability limitations, forcing a trade-off between precision and coverage that reduces its effectiveness for large or compositional contracts.

Dynamic approaches aim to complement static analysis. ReGuard [21] applies fuzzing by translating contracts into C++ and generating randomized transactions to detect reentrancy from execution traces. While effective for certain execution-dependent bugs, fuzzing suffers from limited coverage and difficulty in triggering complex vulnerability conditions.

Runtime monitoring tools such as Sereum [30] and ECFChecker [15] detect reentrancy during execution by tracking control and data flows, achieving high precision but focusing on detection and prevention rather than abstraction and reuse.

Recent work has explored compositional and cross-contract reentrancy detection. Sailfish [6], Pluto [23], and SliSE [42] extend analysis across contract boundaries, improving coverage of complex interactions. Nevertheless, these approaches rely on ad hoc detection rules and lack a unified abstraction that explains different reentrancy behaviors in a systematic manner.

Existing approaches either rely on conservative heuristics, suffer from scalability issues, or focus solely on detection at runtime. None provides a pattern-based characterization of reentrancy grounded in EVM execution semantics, together with precise and reusable detection logic. This paper addresses this gap by defining a design-pattern–based taxonomy of reentrancy vulnerabilities and implementing logic rules for their detection.

4 Reentrancy Vulnerability

Reentrancy is one of the most critical and widely studied vulnerabilities affecting Ethereum smart contracts. It arises when a contract transfers control to an external contract before completing its own state updates, allowing the external contract to call back into the original contract and manipulate its state while it is in an inconsistent intermediate state [2] [30] [34]. The practical severity of reentrancy has been demonstrated by several real-world incidents. The most prominent example is The DAO attack in 2016, which ultimately led to the Ethereum hard fork [2] due to its massive financial and ecosystem impact, with losses estimated at around USD 60 million. The vulnerability resided in the DAO contract's withdrawal function [11], which transferred Ether to the user before updating the user's internal balance. A malicious contract exploited this flaw by repeatedly re-entering the withdrawal function during fallback execution before the balance update took place. Similar reentrancy-based exploits have continued to affect major decentralized finance platforms, including Uniswap [27] and Lendf.Me [10] in April 2020, and BurgerSwap [44] in May 2021, with losses ranging from hundreds of thousands to millions of dollars.

Reentrancy vulnerabilities can arise in several forms depending on how control returns to the vulnerable contract. In *single-function reentrancy* [5], the attacker repeatedly re-enters the same function before its state is updated. In *cross-function reentrancy* [40] [30], the re-entry occurs through a different function within the same contract, affecting shared state in a less obvious way. *Cross-contract reentrancy* [41] involves multiple interacting contracts and more complex call chains. Reentrancy may also occur through `DELEGATECALL` [18], where

external code executes in the caller's storage context, creating risks especially in proxy-based designs. Finally, although less common, reentrancy can also emerge during contract creation through `CREATE` or `CREATE2` [30], when initialization logic enables unexpected callbacks before deployment is fully completed.

Several protection mechanisms have been adopted to mitigate reentrancy vulnerabilities in smart contracts. The most widely recommended practice is the *Checks-Effects-Interactions* (CEI) pattern, which requires contracts to first validate conditions, then update their internal state, and only afterward perform external calls. By ensuring that critical state changes are completed before control is transferred outside the contract, CEI reduces the risk of reentrant execution. Another common defense is the use of a mutex-based guard, usually implemented through a shared lock variable or a `nonReentrant` modifier, which prevents a function from being invoked again before the current execution has finished. Historically, Solidity's `send` and `transfer` primitives were also considered safer because they forward only a limited amount of gas to the callee, thereby restricting reentrant behavior, although this protection is no longer considered fully reliable under evolving gas semantics. Other mechanisms, such as `onlyOwner` access control and `isHuman` checks, may reduce exposure by limiting who can invoke sensitive functions, but they do not provide complete protection against reentrancy, especially in more complex cross-function or cross-contract scenarios.

5 Reentrancy Patterns

In this work, we adopt the definition of a *pattern* proposed by Destefanis et al. [24], who describe a design pattern as "a typical solution to a recurring design problem." The notion of pattern originates from architecture then later transferred to software engineering and became formally established in object-oriented design [14]. A design pattern is generally characterized by three main elements: a *name*, which provides a clear and memorable reference; a *problem*, which defines the recurring issue and the conditions under which the pattern applies; and a *solution*, which presents an abstract but reusable approach for resolving the problem without prescribing a single concrete implementation.

5.1 Baseline Reentrancy Pattern (TxSpector)

We first restate the baseline reentrancy pattern implemented in TxSpector [45] to clarify its detection scope and serve as a reference for the extended patterns introduced later.

The pattern presented in Table 1 is effective for detecting DAO-style reentrancy attacks, as it precisely captures executions in which control-flow decisions rely on stale storage values.

[1] Depth indicates the nesting level of calls, that is, the call-stack level at which an opcode is executed; it increases on external calls and decreases on return.
[2] Call number records the number of calls that have occurred before a given opcode, thereby identifying the call instance to which the corresponding opcode belongs.
[3] Index denotes the position of an opcode in the execution flow graph and determines the execution order of the corresponding instruction.

Table 1 Baseline reentrancy pattern as implemented in TxSpector.

Field	Description
Name	Single-Function Reentrancy
Problem	A function reads a storage variable (e.g., user balance) and later makes an external call. The execution flow of the function depends on the value read from storage. During the external call, a malicious contract re-enters the same function before the original execution completes. As a result, the storage value used in the control-flow decision becomes stale, allowing the attacker to exploit the inconsistent contract state.
Detection Rules	The detection rules are divided into two main conditions. Condition 1 checks the `SLOAD`–`JUMPI` dependency: it verifies whether a value read from storage through `SLOAD` is subsequently used in a control-flow decision, namely as part of the condition of a `JUMPI`. To ensure that both instructions belong to the same execution frame, the `SLOAD` and `JUMPI` must have the same depth[1] and call number[2]. Condition 2 checks the `SLOAD`–`SSTORE` dependency: it first determines whether there is an `SSTORE` that updates the same storage address previously read by the `SLOAD` satisfying Condition 1. It then verifies that the `SSTORE` occurs after the `SLOAD` and after the return from an external call, by checking both index[3] and depth. In particular, the `SLOAD` index must be smaller than the `SSTORE` index, and the `SLOAD` depth must be greater than the `SSTORE` depth. Finally, the rules ensure that `SLOAD`, `SSTORE`, and `JUMPI` all belong to the same contract.

5.2 New Reentrancy Patterns

Table 2 Cross-function reentrancy pattern as implemented in TXSPECTOR.

Field	Description
Name	Cross-Function Reentrancy
Problem	A function reads a storage variable and performs an external call before updating the contract state. During that call, a malicious contract re-enters the same contract through a different function that modifies the same or a related state variable. Since the original execution has not yet completed, the contract may enter an inconsistent state that can be exploited.
Detection Rules	The detection is divided into two phases. **Phase 1** identifies the outer function f_a: it checks whether a function reads a storage variable using `SLOAD`, whether the loaded value is used in the condition of a `JUMPI`, and whether a call-related opcode is subsequently executed. The execution order must satisfy `SLOAD index < JUMPI index < CALL index`, and all three instructions must share the same depth and call number, indicating that they belong to the same execution frame. **Phase 2** identifies the re-entered function f_b: it checks whether another function, with a different function signature from f_a, updates a storage variable using `SSTORE`. It then verifies that f_b and its `SSTORE` execute in the same frame, and that their depth and call number are greater than those of the outer function, indicating re-entry into a deeper call context. In addition, the `SSTORE` must occur after the external call, that is, `SSTORE index > CALL index`. Finally, the rule requires that f_a, f_b, `SLOAD`, `JUMPI`, the call-related opcode, and `SSTORE` all belong to the same smart contract.

We now extend the baseline single-function pattern by introducing new reentrancy patterns that capture beaviors spanning multiple functions within the same contract. In particular, we formalize cross-function reentrancy, whose pattern is presented in Table 2.

The detection rules rely on relations that encode low-level execution events and data dependencies as logical facts. In particular, opcode relations such as `op_SLOAD`, `op_SSTORE`, and `op_JUMPI` capture the execution of specific EVM instructions together with their operands, program location, and execution context, identified by the call number and call depth. For example, `op_SLOAD`(_, *sload_addr_var*, *sload_val*, *sload_loc*, *depthA*, *cnA*) denotes an `SLOAD` instruction where the first argument is an unused parameter, *sload_addr_var* represents the storage address being accessed, *sload_val* is the value loaded from storage, *sload_loc* is the program location of the instruction, and *depthA* and *cnA* identify the depth and the call number of that opcode. Other relations capture higher-level execution properties. `FunctionSelector` identifies the selector of the function executed in a given frame, `CallOperator` represents external call instructions that may transfer control outside the contract, and `depends` expresses a data dependency between two variables. These relations are encoded in Datalog [1] as follows:

Listing 1 Auxiliary relations.

```
CallOperator(target_val, call_loc, call_cd, call_cn) :-
  (op_CALL(_, _, target_var, _, _, _, _, _, _, call_loc, call_cd, call_cn);
   op_STATICCALL(_, _, target_var, _, _, _, _, _, _, call_loc, call_cd, call_cn);
   op_DELEGATE(_, _, target_var, _, _, _, _, _, _, call_loc, call_cd, call_cn);
   op_CALLCODE(_, _, target_var, _, _, _, _, _, _, call_loc, call_cd, call_cn)),
  value(target_var, target_val).

// Extract function selector from CALLDATALOAD(0)
// The selector is the first 4 bytes of calldata (located at offset 0)
.decl FunctionSelector(cn:number, depth:number, loc:number, selector:Variable)

FunctionSelector(cn, depth, loc, selector) :-
  op_CALLDATALOAD(_, index_var, selector, loc, depth, cn),
  value(index_var, "0x0").
```

Listing 2 Detection rule for the outer function.

```
.decl OuterFunction(
  cnA:number, depthA:number, selA:Variable,
  sload_addr_var:Variable, sload_val:Variable, sload_loc:number,
  jumpi_cond:Variable, jumpi_loc:number,
  call_loc:number
)

OuterFunction(cnA, depthA, selA, sload_addr_var, sload_val, sload_loc, jumpi_cond,
    jumpi_loc, call_loc) :-
  // Function A selector
  FunctionSelector(cnA, depthA, sel_loc, selA),
  // SLOAD in Function A - reads state
  op_SLOAD(_, sload_addr_var, sload_val, sload_loc, depthA, cnA),
  // JUMPI in Function A that depends on the SLOAD value
  op_JUMPI(_, _, jumpi_cond, jumpi_loc, depthA, cnA),
  depends(jumpi_cond, sload_val),
  // External call in Function A
  CallOperator(_, call_loc, depthA, cnA),
  // Ordering
  sel_loc < sload_loc,
  sload_loc < jumpi_loc,
  jumpi_loc < call_loc.
```

Listing 3 Detection rule for the re-entered function.

```
.decl ReenteredFunction(
  cnB:number, depthB:number, selB:Variable,
  sstore_addr_var:Variable, sstore_loc:number
)

ReenteredFunction(cnB, depthB, selB, sstore_addr_var, sstore_loc) :-
  // Function B selector
  FunctionSelector(cnB, depthB, sel_b_loc, selB),
  // SSTORE in Function B - modifies state
  op_SSTORE(_, sstore_addr_var, _, sstore_loc, depthB, cnB).
```

Listing 4 Detection rule for cross-function reentrancy.

```
.decl CrossFunctionReentrancy(
  cnA:number, depthA:number, selA:Variable,
  cnB:number, depthB:number, selB:Variable,
  sload_loc:number, sstore_loc:number,
  jumpi_loc:number, call_loc:number,
  sload_storage_addr:Value, sstore_storage_addr:Value
)
.output CrossFunctionReentrancy

CrossFunctionReentrancy(cnA, depthA, selA, cnB, depthB, selB, sload_loc, sstore_loc,
     jumpi_loc, call_loc, sload_storage_addr, sstore_storage_addr) :-
  // Outer function pattern
  OuterFunction(cnA, depthA, selA, sload_addr_var, sload_val, sload_loc, jumpi_cond,
     jumpi_loc, call_loc),
  // Re-entered function pattern
  ReenteredFunction(cnB, depthB, selB, sstore_addr_var, sstore_loc),
  // Reentrancy condition
  cnB > cnA,
  depthB > depthA,
  // Different functions
  selA != selB,
  // Temporal ordering
  sload_loc < sstore_loc,
  // State modification after external call
  sstore_loc > call_loc.
```

5.3 Build Variable Dependency Graph

Algorithm 1 constructs a variable dependency graph to determine whether state variables read via `SLOAD` influence control flow decisions. The algorithm computes the transitive closure of definition-use chains by establishing reflexive dependencies, identifying direct dependencies within statements, and then computing transitive dependencies through intermediate variables.

5.4 Extract Function Selectors

Algorithm 2 extracts function selectors by identifying `CALLDATALOAD` operations that read from offset 0, which contains the 4-byte function selector. For traces without explicit function dispatch, it falls back to using the call context $(cn, depth)$ as a function identifier.

Algorithm 1 Build Variable Dependency Graph.

Require: Execution trace T
Ensure: Dependency relation $D : \mathit{Variable} \times \mathit{Variable} \to \{\mathit{true}, \mathit{false}\}$
  $D \leftarrow \emptyset$
  // **Reflexive dependencies**
  for each variable x in T **do**
    $D(x, x) \leftarrow \mathit{true}$
  end for
  // **Direct dependencies (def-use chains)**
  for each statement stmt in T **do**
    for each $\mathit{def}(x, \mathit{stmt}, \ldots)$ and $\mathit{use}(y, \mathit{stmt}, \ldots)$ **do**
      $D(x, y) \leftarrow \mathit{true}$
    end for
  end for
  // **Transitive closure**
  repeat
    $\mathit{changed} \leftarrow \mathit{false}$
    for each variables x, y, z **do**
      if $D(x, y) \wedge D(y, z) \wedge \neg D(x, z)$ **then**
        $D(x, z) \leftarrow \mathit{true}$
        $\mathit{changed} \leftarrow \mathit{true}$
      end if
    end for
  until $\neg \mathit{changed}$
  return D

Algorithm 2 Extract Function Selectors.

Require: Execution trace T
Ensure: Function mapping $F : (\mathit{CallNumber}, \mathit{Depth}) \to \mathit{Selector}$
  $F \leftarrow \emptyset$
  for each $\mathit{op\_CALLDATALOAD}(\mathit{index}, \mathit{result}, \mathit{loc}, \mathit{depth}, \mathit{cn})$ in T **do**
    if $\mathit{value}(\mathit{index}) = 0x0$ **then** ▷ First 4 bytes of calldata
      $F(\mathit{cn}, \mathit{depth}) \leftarrow \mathit{value}(\mathit{result})$
    end if
  end for
  // **Fallback: use call context as identifier**
  for each $\mathit{op\_SLOAD}$ or $\mathit{op\_SSTORE}$ at $(\mathit{loc}, \mathit{depth}, \mathit{cn})$ **do**
    if $F(\mathit{cn}, \mathit{depth})$ is undefined **then**
      $F(\mathit{cn}, \mathit{depth}) \leftarrow (\mathit{cn}, \mathit{depth})$ ▷ Use context as identifier
    end if
  end for
  return F

5.5 Cross-Function Reentrancy Detection

Algorithm 3 identifies cross-function reentrancy by matching two function patterns: Function A reads state via `SLOAD`, uses it in a conditional jump, and makes an external call; Function B modifies state via `SSTORE`. A vulnerability is detected when both functions are different, Function B is re-entered after Function A's call (indicated by increased call number and depth), and temporal ordering is satisfied (read before write, write after call).

Algorithm 3 Cross-Function Reentrancy Detection.

Require: Execution trace $T = \{op_1, op_2, \ldots, op_n\}$ with metadata $(loc, cn, depth, vars)$
Ensure: Set of cross-function reentrancy vulnerabilities R

```
// Step 1: Build variable dependency graph
D ← BuildDependencyGraph(T)
// Step 2: Extract function selectors
F ← ExtractFunctionSelectors(T)
// Step 3: Identify Function A patterns
A ← ∅
for each op_SLOAD(addr_A, val_A, loc_sload, d_A, cn_A) in T do
    sel_A ← F(cn_A, d_A)
    if ∃op_JUMPI(cond, loc_jumpi, d_A, cn_A) where D(cond, val_A) then
        if ∃CallOperator(target, loc_call, d_A, cn_A) where loc_sload < loc_jumpi < loc_call then
            A ← A ∪ {(cn_A, d_A, sel_A, addr_A, loc_sload, loc_jumpi, loc_call)}
        end if
    end if
end for
// Step 4: Identify Function B patterns
B ← ∅
for each op_SSTORE(addr_B, val_B, loc_sstore, d_B, cn_B) in T do
    sel_B ← F(cn_B, d_B)
    B ← B ∪ {(cn_B, d_B, sel_B, addr_B, loc_sstore)}
end for
// Step 5: Detect cross-function reentrancy
R ← ∅
for each (cn_A, d_A, sel_A, addr_A, loc_sload, loc_jumpi, loc_call) ∈ A do
    for each (cn_B, d_B, sel_B, addr_B, loc_sstore) ∈ B do
        if sel_A ≠ sel_B then                                   ▷ Different functions
            if cn_B > cn_A ∧ d_B > d_A then                      ▷ Reentrancy condition
                if loc_sload < loc_sstore ∧ loc_sstore > loc_call then   ▷ Temporal ordering
                    R ← R ∪
                    {(cn_A, d_A, sel_A, cn_B, d_B, sel_B, loc_sload, loc_sstore, loc_jumpi, loc_call, value(addr_A))}
                end if
            end if
        end if
    end for
end for
return R
```

6 Implementation

We evaluate our rules by implementing and testing them within TxSpector, which consists of four main components: the Trace Extractor, the Execution Flow Graph Generator, the Logic Relation Builder, and the Attack Detector. Our contribution mainly concerns the Trace Extractor and the Detector.

For the Trace Extractor, which replays Ethereum on-chain transactions using a Geth node and records execution traces, we found that the Geth version originally used by TxSpector was outdated and did not support several newer EVM opcodes. In addition, some of the libraries required for extraction were no longer available. To address this limitation, we migrated the extractor to a newer version of Geth and applied the necessary modifications to restore compatibility with TxSpector while supporting recent EVM changes. These updates required modifications to several source files responsible for MongoDB initialization and transaction storage, transaction execution, state prefetching, EVM interpretation, opcode execution, and trace persistence.

We also updated the Detector component so that it recognizes newer call-related opcodes during the analysis, and we extended it with new user-defined Datalog rules for detecting cross-function reentrancy. We then evaluated the extended tool on real Ethereum on-chain transactions. The following example was successfully detected by our approach. Figure 1 illustrates the process flow of the detector component. The complete implementation of these modifications along the examples used to validate the proposed detection rules, is publicly available on GitHub.

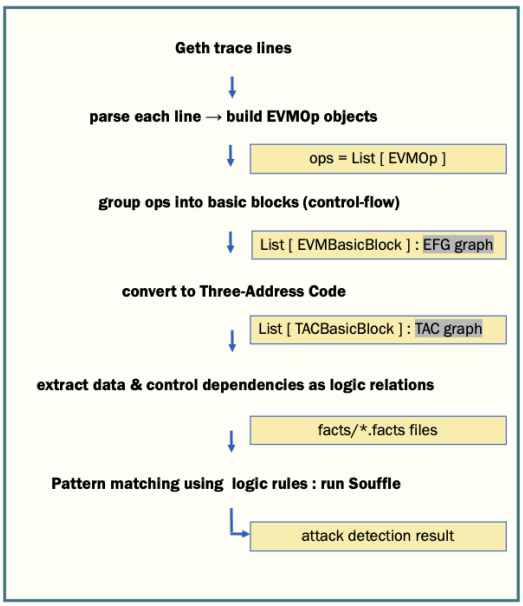

Figure 1 Process flow of the TXSpector detector component.

6.1 Example: Detected Cross-Function Reentrancy

To illustrate the detection algorithm, we present a concrete example of a cross-function reentrancy vulnerability identified in our analysis. Table 3 shows the raw output from the detection system.

Table 3 reports the output produced by our detection rule for a cross-function reentrancy instance. The result shows that the outer function f_a, executed with call number 2 at depth 2, reads a storage value through `SLOAD` at program location 666 and later performs an external call at location 732. During this external interaction, a second function f_b, executed with call number 3 at depth 3, is re-entered and writes to storage through `SSTORE` at location 771.

Table 3 Detection output for cross-function reentrancy. **cnA** and **cnB** denote the call numbers of the outer function f_a and the re-entered function f_b, while **dA** and **dB** represent their call depths in the execution stack. **sload_idx**, **sstore_idx**, and **call_idx** indicate the program locations of the corresponding EVM instructions. **Contract** denotes the address of the analyzed smart contract. **Sel. A** and **Sel. B** represent the function selectors identifying f_a and f_b, respectively. A function selector corresponds to the first four bytes of the Keccak-256 hash of the function signature.

cnA	dA	cnB	dB	sload_idx	sstore_idx	call_idx	Contract	Sel. A	Sel. B
2	2	3	3	666	771	732	`0xBf4e...ca754`	`0xa..cbb`	`0xb..a905`

This execution is classified as cross-function reentrancy because all required conditions are satisfied. First, the reentrancy condition holds, since the second function executes in a deeper call context and with a higher call number than the first one, that is, $3 > 2$ for both call number and depth. Second, both functions access the same storage address, indicating that the state modified by f_b is the same state previously read by f_a. Third, the temporal ordering is consistent with a reentrancy scenario: the storage read occurs before the storage write ($666 < 771$), and the storage write occurs after the external call ($771 > 732$).

Overall, this output illustrates the characteristic pattern of cross-function reentrancy: one function reads a state variable and performs an external call before completing its execution, while a different re-entered function modifies that same state variable during the intermediate execution window. This may invalidate the assumptions made by the outer function and lead to exploitable inconsistent behavior.

7 Discussion

This work shows that reentrancy vulnerabilities cannot be fully captured by traditional single-function or source-level analyses. By operating directly on EVM transaction traces, our approach characterizes reentrancy as an execution-level phenomenon arising from concrete control-flow and data-dependency relations. This perspective enables precise reasoning about behaviors that depend on call context, temporal ordering, and shared persistent state.

A key contribution is the formalization and detection of cross-function reentrancy, where re-entry occurs through a different function of the same contract. Such vulnerabilities are missed by existing tools that assume reentrancy to be confined to a single function. By explicitly modeling call depth, call number, and storage access ordering, our logic rules accurately identify these behaviors without relying on conservative heuristics, thereby reducing false positives.

The pattern-based abstraction adopted in this work provides a structured and extensible framework for reentrancy detection. New variants can be incorporated by defining additional patterns and logic rules, without modifying the underlying trace extraction pipeline. While the approach is inherently limited by trace coverage, it complements static and symbolic techniques and is well suited for analyzing real-world transaction executions.

8 Conclusion

This paper presented a pattern-based approach for detecting reentrancy vulnerabilities at the level of EVM execution traces. We extended an existing execution-trace analysis framework with new logic rules designed to capture cross-function reentrancy, a more subtle and dangerous variant of the classical reentrancy vulnerability. Unlike traditional analyses that focus primarily on single-function patterns or source-level code, our approach operates

directly on execution traces and models reentrancy as a runtime phenomenon involving control-flow dependencies, storage access ordering, and call-context relations.

To enable this analysis, we modernized the underlying infrastructure by updating its trace extraction component to support recent versions of the Ethereum client and newer EVM instructions. We then implemented the proposed detection rules in Datalog within the detector module and evaluated the extended system on real Ethereum transaction traces. The experimental results demonstrate that our approach successfully identifies cross-function reentrancy patterns that are not captured by the baseline rules.

Beyond the specific case of reentrancy, the pattern-based methodology proposed in this work provides a flexible framework for describing and detecting execution-level vulnerabilities in smart contracts. Future work includes extending the pattern taxonomy to cover additional classes of vulnerabilities, improving trace coverage, and integrating the approach with complementary static and symbolic analysis techniques to further enhance detection accuracy.

References

1 Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

2 Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In *Principles of Security and Trust*, volume 10204 of *LNCS*, pages 164–186. Springer, 2017. doi:10.1007/978-3-662-54455-6_8.

3 Syed Badruddoja, Ram Dantu, Yanyan He, Kritagya Upadhayay, and Mark Thompson. Making smart contracts smarter. In *2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–3, 2021. doi:10.1109/ICBC51069.2021.9461148.

4 Massimo Bartoletti, Lorenzo Benetollo, Michele Bugliesi, Silvia Crafa, Giacomo Dal Sasso, Roberto Pettinau, Andrea Pinna, Mattia Piras, Sabina Rossi, Stefano Salis, Alvise Spanò, Viacheslav Tkachenko, Roberto Tonelli, and Roberto Zunino. Smart contract languages: A comparative analysis. *Future Gener. Comput. Syst.*, 164:107563, 2025. doi:10.1016/j.future.2024.107563.

5 Lorenzo Benetollo, Semia Guesmi, Carla Piazza, Dalila Ressi, Sabina Rossi, and Alvise Spanò. Modeling reentrancy in smart contracts through noninterference. In *Proceedings of the Seventh Distributed Ledger Technology Workshop (DLT 2025), Pizzo, Italy, June 12-14 2025*, volume 4105 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2025. URL: https://ceur-ws.org/Vol-4105/paper11.pdf.

6 Priyanka Bose, Dipanjan Das, Yanju Chen, Yu Feng, Christopher Kruegel, and Giovanni Vigna. Sailfish: Vetting smart contract state-inconsistency bugs in seconds. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 161–178, 2022. doi:10.1109/SP46214.2022.9833721.

7 Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. Vandal: A scalable security analysis framework for smart contracts, 2018. arXiv:1809.03981.

8 Michele Bugliesi, Lucia Gallina, Andrea Marin, Sabina Rossi, and Sardaouna Hamadou. Interference-sensitive preorders for manets. In *Ninth International Conference on Quantitative Evaluation of Systems, QEST 2012, London, United Kingdom, September 17-20, 2012*, pages 189–198. IEEE Computer Society, 2012. doi:10.1109/QEST.2012.15.

9 ConsenSys Diligence. Mythril: Security analysis tool for ethereum smart contracts. https://github.com/ConsenSysDiligence/mythril, 2024. Accessed: January 2026.

10 K. Davidoz. The reentrancy strikes again: The case of lendf.me, 2020. Accessed: 2026-03-15. URL: https://medium.com/@kdavidoz/the-reentrancy-strikes-again-the-case-of-lendf-me-4359016776f3.

11 Etherscan. The dao smart contract source code. https://etherscan.io/address/0xbb9bc244d798123fde783fcc1c72d3bb8c189413#code, 2016. Accessed: 2026-03-15.

12 Josselin Feist, Gustavo Grieco, and Alex Groce. Slither: A static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 8–15. IEEE, May 2019. doi:10.1109/wetseb.2019.00008.

13 Lucia Gallina, Sardaouna Hamadou, Andrea Marin, and Sabina Rossi. A probabilistic energy-aware model for mobile ad-hoc networks. In *Analytical and Stochastic Modeling Techniques and Applications - 18th International Conference, ASMTA 2011, Venice, Italy, June 20-22, 2011. Proceedings*, volume 6751 of *Lecture Notes in Computer Science*, pages 316–330. Springer, 2011. doi:10.1007/978-3-642-21713-5_23.

14 Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

15 Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. Online detection of effectively callback free objects with applications to smart contracts, 2018. arXiv:1801.04032.

16 Samia Guesmi, Carla Piazza, and Sabina Rossi. Noninterference analysis for smart contracts: Would you bet on it? In *Proceedings of the Sixth Distributed Ledger Technology Workshop (DLT 2024), Turin, Italy, May 14-15, 2024*, volume 3791 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2024. URL: https://ceur-ws.org/Vol-3791/paper12.pdf.

17 Bo Jiang, Ye Liu, and Wing Kwong Chan. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, pages 259–269, 2018. doi:10.1145/3238147.3238177.

18 Jiao Jiao, Shuanglong Kan, Shang-Wei Lin, David Sanan, Yang Liu, and Jun Sun. Semantic understanding of smart contracts: Executable operational semantics of solidity. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1695–1712, 2020. doi:10.1109/SP40000.2020.00066.

19 Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. Zeus: Analyzing safety of smart contracts. In *Network and Distributed System Security Symposium*, January 2018. doi:10.14722/ndss.2018.23082.

20 Zeqin Liao, Zibin Zheng, Xiao Chen, and Yuhong Nan. Smartdagger: a bytecode-based static analysis approach for detecting cross-contract vulnerability. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 752–764, July 2022. doi:10.1145/3533767.3534222.

21 Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. Reguard: Finding reentrancy bugs in smart contracts. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 65–68, 2018.

22 Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 254–269, October 2016. doi:10.1145/2976749.2978309.

23 Fuchen Ma, Zhenyang Xu, Meng Ren, Zijing Yin, Yuanliang Chen, Lei Qiao, Bin Gu, Huizhong Li, Yu Jiang, and Jiaguang Sun. Pluto: Exposing vulnerabilities in inter-contract scenarios. *IEEE Transactions on Software Engineering*, 48(11):4380–4396, 2022. doi:10.1109/TSE.2021.3117966.

24 Lodovica Marchesi, Michele Marchesi, Giuseppe Destefanis, Giulio Barabino, and Danilo Tigano. Design patterns for gas optimization in ethereum. In *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 9–15, 2020. doi:10.1109/IWBOSE50093.2020.9050163.

25 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Cryptography Mailing list at https://metzdowd.com*, March 2009.

26 Tai D. Nguyen, Long H. Pham, Jun Sun, Yun Lin, and Quang Tran Minh. sfuzz: An efficient adaptive fuzzer for solidity smart contracts, 2020. arXiv:2004.08563.

27 OpenZeppelin. Exploiting uniswap: From reentrancy to actual profit, 2019. Accessed: 2026-03-15. URL: https://www.openzeppelin.com/news/exploiting-uniswap-from-reentrancy-to-actual-profit.

28 OSU Security Lab. Txspector: Uncovering attacks in ethereum from transactions, 2020. URL: https://github.com/OSUSecLab/TxSpector.

29 Heidelinde Rameder, Monika di Angelo, and Gernot Salzer. Review of automated vulnerability analysis of smart contracts on ethereum. *Frontiers in Blockchain*, Volume 5 - 2022, 2022. doi:10.3389/fbloc.2022.814977.

30 Michael Rodler, Wenting Li, Ghassan O. Karame, and Lucas Davi. Sereum: Protecting existing smart contracts against re-entrancy attacks, 2018. arXiv:1812.05934.

31 Noama Fatima Samreen and Manar H. Alalfi. A survey of security vulnerabilities in ethereum smart contracts, 2021. arXiv:2105.06974.

32 Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. ethor: Practical and provably sound static analysis of ethereum smart contracts, 2020. arXiv:2005.06227.

33 Nipun Sharma, Swati Sharma, et al. A survey of mythril, a smart contract security analysis tool for evm bytecode. *Indian Journal of Natural Sciences*, 13(75):51003–51010, 2022.

34 Solidity Team. Solidity documentation: Security considerations. https://docs.soliditylang.org/en/latest/security-considerations.html. Accessed 2026-03-11.

35 Solidity Team. Solidity programming language. https://www.soliditylang.org, 2024. Accessed: January 2026.

36 Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. Smartcheck: Static analysis of ethereum smart contracts. In *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 9–16, 2018.

37 Christof Ferreira Torres, Antonio Ken Iannillo, Arthur Gervais, and Radu State. Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts, 2021. arXiv:2005.12156.

38 Trail of Bits. Manticore: Symbolic execution tool for smart contract and binary analysis. https://github.com/trailofbits/manticore, 2024. Accessed: January 2026.

39 Petar Tsankov, Andrei Dan, Dana Drachsler Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. Securify: Practical security analysis of smart contracts, 2018. arXiv:1806.01143.

40 Valix Consulting. Solidity smart contract security by example #04: Cross-function reentrancy, 2019. Medium article, accessed: 2026-03-15. URL: https://medium.com/valixconsulting/solidity-smart-contract-security-by-example-04-cross-function-reentrancy-de9cbce0558e.

41 Valix Consulting. Solidity smart contract security by example #05: Cross-contract reentrancy. Medium, 2019. Accessed: 2026-03-15. URL: https://medium.com/valixconsulting/solidity-smart-contract-security-by-example-05-cross-contract-reentrancy-30f29e2a01b9.

42 Zexu Wang, Jiachi Chen, Yanlin Wang, Yu Zhang, Weizhe Zhang, and Zibin Zheng. Efficiently detecting reentrancy vulnerabilities in complex smart contracts, 2024. doi:10.48550/arXiv.2403.11254.

43 Jiaming Ye, Mingliang Ma, Yun Lin, Yulei Sui, and Yinxing Xue. Clairvoyance: cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, pages 274–275, June 2020. doi:10.1145/3377812.3390908.

44 ZenGo. Burgerswap vulnerability analysis, 2021. Accessed: 2026-03-15. URL: https://zengo.com/burgerswap-vulnerability/.

45 Mengya Zhang, Xiaokuan Zhang, Yinqian Zhang, and Zhiqiang Lin. TXSPECTOR: Uncovering attacks in ethereum from transactions. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2775–2792. USENIX Association, August 2020. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/zhang-mengya.