Confusum Contractum:
Confused Deputy Vulnerabilities in Ethereum Smart Contracts

Abstract

Smart contracts are immutable programs executed in the context of a globally distributed system known as a *blockchain*. They enable the decentralized implementation of many interesting applications, such as financial protocols, voting systems, and supply-chain management. In many cases, multiple smart contracts need to work together and communicate with one another to implement complex business logic. However, these smart contracts must take special care to guard against malicious interactions that might lead to the violation of a contract's security properties and possibly result in substantial financial losses.

In this paper, we introduce a class of inter-program communication flaws that we call *confused contract vulnerabilities*. This type of bug is an instance of the *confused deputy* vulnerability, set in the new context of smart contract intercommunication. When exploiting a confused contract bug, an attacker is able to divert a remote (inter-contract) call in a confused (victim) contract to a target contract and function of the attacker's choosing. The call performs sensitive operations on behalf of the confused contract, which can result in financial loss or malicious modifications of the persistent storage of the involved contracts.

To identify opportunities for confused contract attacks at scale, we implemented KAI, a system that is able to automatically identify confused contracts and candidate target contracts on the Ethereum mainnet. We leveraged KAI to analyze a total of 2,335,193 smart contracts deployed in the past two years, and we identified 529 potential confused contracts. When investigating these warnings, we discovered past and present opportunities for confused contract attacks that could have compromised digital assets worth more than one million US dollars.

1 Introduction

In the past few years, the interest in blockchain-powered decentralized applications (dApps) has risen considerably. This interest drove the market capitalization of Ethereum [17] – the most popular blockchain platform for dApps – from 51 billion to 568 billion dollars in just one year (Nov. 2020 - Nov. 2021) [69]. Despite the recent market correction, which saw Ethereum's market cap drop to 201 billion dollars [57], the ecosystem is still able to attract huge investments. One reason is the broad interest in and development of many types of distributed applications.

Blockchain-powered applications are implemented via *smart contracts*: immutable programs stored on the blockchain in the form of bytecode, which is executed by the Ethereum Virtual Machine (EVM) [65]. Users interact with the system via signed *transactions*, wherein they specify a smart contract's function they want to invoke, together with its arguments. The Ethereum network's distributed consensus mechanism produces a strict ordering of transactions, which are emitted at regular intervals ($\sim$12 seconds) in groups called *blocks*. All participating nodes in the network execute each transaction in order whenever a new block is created. Through this process, the Ethereum network maintains a consistent global state across all nodes.

Smart contract functions commonly manipulate financial assets, such as tokens and Ether (ETH) [18], which is the native cryptocurrency on the Ethereum network. Smart contracts may also read and write their own persistent storage, emit log events, and call functions of other contracts. Through their functionality, smart contracts help to build the Decentralized Finance (DeFi) [16] ecosystem: a complex network of decentralized financial protocols enabled by the blockchain infrastructure.

In contrast to the traditional financial system, DeFi promises to bring an increase in both transparency and democratization of financial tools, together with full accessibility and control over personal funds [49]. However, DeFi's benefits come at the cost of drastically increasing the risk of financial losses. In fact, as smart contracts run on a public blockchain, they can interact with – and be scrutinized by – anyone. In particular, when a smart contract suffers from a security bug, a successful attack from a malicious actor can potentially drain

millions of dollars worth of assets. It can be very hard, if not impossible, to recover lost funds due to the immutability of the blockchain network [50] and the intrinsic lack of a central authority. The higher stakes for blockchain security attract a multitude of different kinds of actors, who race toward discovering security vulnerabilities in smart contracts. As a result, a number of multi-million-dollar bug bounties were recently awarded to white hat hackers [9].

Thanks to the attention from both security professionals and the academic research community, the impact and prevalence of certain classes of vulnerabilities has been considerably reduced in modern contracts. This includes *integer overflow bugs*, via the usage of safe math libraries [46], and *re-entrancy* [6], via specific checks ensuring single entrance [45]. In addition, these and other "traditional" vulnerabilities have been the target of many automated verification and bug-finding solutions [3, 4, 10, 25, 27, 31, 32, 39, 47, 52].

Interestingly, recent attacks made use of cross-contract vulnerabilities that have little to no automated discovery support, and they can be difficult to identify even via manual reviews performed by experts. An important property of smart contracts is the fact that their public functions can be directly called by *any* other programs on the blockchain at *any* point in time. Hence, it is common for smart contracts to guard sensitive functionality with access control checks that are designed to accept interactions only from specific callers (these can be user accounts or contracts). Unfortunately, the complicated mix of inter-contract communication, and the often convoluted and custom access control policies implemented by smart contracts, can easily result in hidden logic bugs. An example of such a bug is an inter-contract vulnerability that affected one of the biggest DeFi platforms, PolyNetwork [48], and led to a financial loss of $610 million in August 2021 [51]. In this case, the attacker exploited an inter-contract communication bug in one of the primary PolyNetwork contracts, misdirecting one of its remote calls to another PolyNetwork contract that is responsible for maintaining the list of active administrators. Then, thanks to the existing trust relationship between these two contracts, the attacker managed to add themselves as a new administrator, escalating their privileges, and ultimately draining a significant amount of funds.

The complexity, and the impact, of the PolyNetwork attack, certainly calls for a deeper understanding of the roots of cross-contract vulnerabilities and the development of automated solutions that are not only able to identify bugs in a single smart contract but rather find unsafe cross-contract interactions across the entire ecosystem. In this paper, we do just that. First, we introduce and characterize *confused contracts*, an important class of cross-contract vulnerabilities. Then, we present KAI, which is a first step in automatically detecting such flaws.

In particular, in this paper, we make the following contributions:

- We describe the fundamental mechanics at the basis of the *confused contract* class of bugs, which is an instance of the confused deputy class of problems in the context of inter-contract interaction.
- We propose a novel methodology to detect confused contract attacks at scale, and we implement a prototype system that we used to analyze 2,335,193 smart contracts binaries, finding a total of 529 potential vulnerabilities.
- We further investigate a subset of the warnings raised by KAI and confirm them by developing working exploits that have the potential to jeopardize assets worth more than one million US dollars.

2 Background

2.1 Blockchain

A blockchain is a decentralized, distributed ledger on which the participating nodes collectively advance the state of an append-only database. In particular, nodes record transactions over the network and register the creation of new *blocks* in the database. New data can only be added to the end of the blockchain in the form of a new block that contains an ordered record of recent transactions. In turn, this creates a permanent, tamper-evident history of all the transactions in the network, allowing for a secure and transparent way to store and transfer data or value. To motivate individuals to actively contribute to the maintenance of the blockchain, a set amount of cryptocurrency is awarded for creating each new block.

2.2 Smart Contracts

Modern blockchains, such as the Ethereum blockchain [17], are known to be "programmable blockchains." Specifically, together with the standard currency-bearing user accounts [19], this kind of blockchain supports the development of decentralized applications (dApps) that can be used to create new kinds of online services. From an implementation point of view, dApps are created by developing *smart contracts*. The code of a smart contract defines its business logic and implements the terms of an agreement between different parties. For instance, a smart contract that sells digital assets might require a specific amount of cryptocurrency (e.g., ETH [18]) to be deposited by a user before the user can register their ownership of an asset. Smart contracts are usually developed using high-level programming languages (e.g., `Solidity` [55], or `Vyper` [61]). Once compiled, smart contracts are stored on-chain in the form of bytecode and executed on-demand by the Ethereum Virtual Machine (EVM).

2.3 Smart Contract Execution

While the execution state of a smart contract is defined by many elements [30], in this paper, we focus on the following four components: (1) program counter, (2) stack, (3) memory, and (4) persistent storage.

Program Counter: The program counter keeps track of the next instruction that needs to be executed.

Stack: The EVM is a stack-based virtual machine, that is, values are pushed and popped onto the stack to perform all arithmetic and control-transfer operations inside the contract.

Memory: The memory is a byte-addressable volatile storage for various EVM instructions. Similar to the stack, the memory starts empty when a smart contract begins its execution. As the code is running, instructions can read and write data at different offsets in memory.

Persistent Storage: The storage is a key-value store with 256-bit keys and 256-bit values. The content of the storage is kept on the blockchain, and hence, persists across multiple smart contract executions. Similarly to memory, specific opcodes (i.e., `SLOAD`, `SSTORE` [65]) can read-/write data at different key slots in the storage.

2.4 Smart Contract Invocations

Any function of a smart contract that is explicitly marked as `public` by the developer is a possible entry point for that contract. That is, it can be directly invoked by any blockchain user who sends a corresponding *transaction* (`Tx`), or another contract that sends an *internal transaction* (`iTx`). When submitting a transaction that invokes the code of a contract, one has to pay a fee (known as the *gas fee*), which is subtracted from the balance of the transaction initiator.

Each transaction has a *context*. Such context includes the `msg.sender` (determining the current address interacting with a contract), the `msg.value` (specifying the amount of ETH transferred), and the `tx.origin` (specifying the sender of the initiating transaction [15]).

The execution of a smart contract in the EVM is quasi-Turing-complete: the EVM can perform any computation as long as the user initiating the execution has sufficient ETH [18] to pay the required gas fee (note that there is an upper limit to the gas fee that can be specified [20]). This mechanism is a defense against denial of service and general abuse of resources [8].

Each transaction includes a byte-string known as the `CALLDATA`. The `CALLDATA` consists of (1) the first four bytes, which identify the `targetFunction` to be executed, and (2) the remaining bytes, which specify the arguments passed to the function. The four bytes that select the target function correspond to the `KECCAK256` [64] hash of the function prototype.

Once a smart contract receives a transaction, the `targetFunction` bytes are extracted from the `CALLDATA` and used to dispatch the execution to the specified function. The function body is then responsible for interpreting the argument data. The computation can either complete successfully (i.e., the execution reaches the end of the function with no error) or fail, resulting in a "revert" of the execution. If the execution is reverted, any changes to the contract's persistent storage are rolled back.

During its execution, a smart contract can invoke functions of other smart contracts, whose code will be executed as part of the *same* transaction. This type of composable software design enables the implementation of advanced application protocols. The communication between different smart contracts happens via so-called *internal transactions*, triggered by one of four EVM opcodes, namely `CALL`, `DELEGATECALL`, `CALLCODE`, and `STATICCALL`. These four opcodes differ in terms of how the caller and callee interact with each other, and how the call's metadata is propagated.

To understand the fundamentals of the *confused contract* vulnerability, it is necessary to discuss the effects of the different opcodes on the involved contracts (sender and receiver) and their execution context. In particular, we look at how caller information is propagated from the sender to the receiver, and whose persistent storage is accessed when the receiver is executing code that includes `SLOAD`/`SSTORE` opcodes. In Figure 1, we summarize the main differences between the execution models of these opcodes and the metadata propagation during the internal transaction (`iTx`). One important difference between each opcode is how persistent storage access is handled. When Contract B uses `CALL` or `STATICCALL` to call a function in Contract C, the receiver (target contract C) accesses its own persistent storage when executing `SLOAD`/`SSTORE` opcodes; in the case of `STATICCALL`, the persistent storage is read-only. On the other hand, when Contract B (the sender) uses either `CALLCODE` or `DELEGATECALL`, a persistent storage operation performed by the code in Contract C accesses the persistent storage of Contract B. A second difference is how `msg.sender` is handled. Specifically, `CALL` and `CALLCODE` change the `msg.sender` attribute of the transaction `iTx` to the address of the sender `B`. For the other two opcodes, the `msg.sender` attribute remains unchanged and holds the value of the originator of the transaction (which is User A in our example).

All four opcodes that are used for smart contract communication require specific arguments [65] to be pushed on the stack before the call:

gas: monetary fee for the execution of the callee's code.

tAddr: the address of the callee.

value: amount of ETH to be sent together with the call (only for `CALL`/`CALLCODE`).

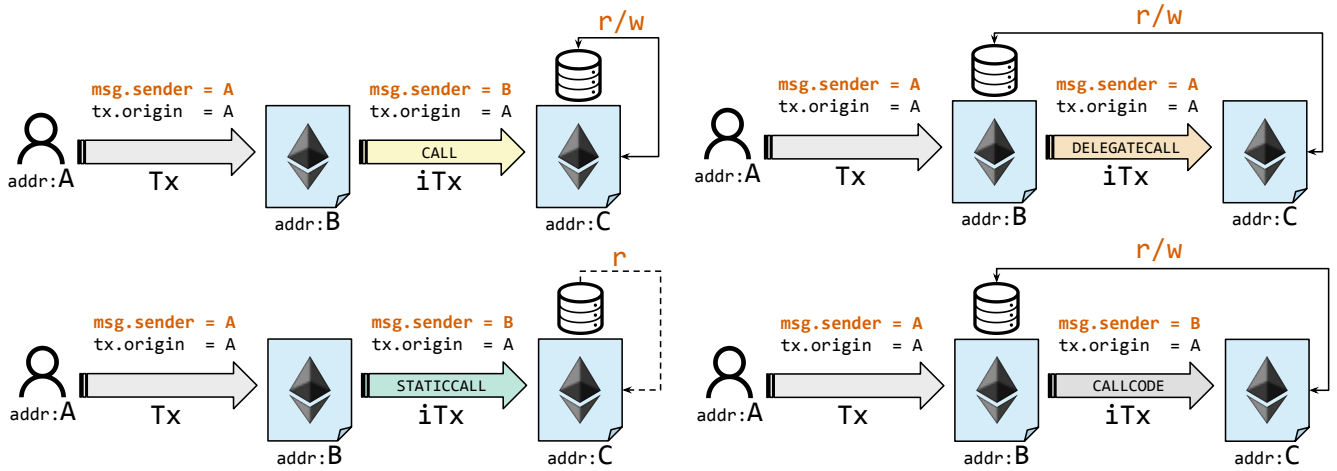

Figure 1: Overview of inter-contract communication operands and their semantics. `Tx` refers to the transaction initiated by a user account (this is `A` in our example). `iTx` refers to an *internal transaction* between smart contracts. We highlighted in red the elements that are relevant to the *confused contract* attack.

argOffset: byte offset in memory where the `CALLDATA` for the `iTx` is expected to be located. The first four bytes specify the `tFunc` signature, followed by the arguments for the function.

argSize: size of the `CALLDATA` located at `tAddr`.

retOffset: byte offset in the memory where the return data computed by the callee will be stored.

retSize: size of the return data computed by the callee.

In this work, we focus on the following key parts of inter-contract communication: the transaction's input, i.e., `CALLDATA`, the target address `tAddr`, and the target function `tFunc`.

3 Motivation and Threat Model

A *confused deputy* vulnerability arises when a higher-privileged component (called the deputy) in a system incorrectly allows a lesser-privileged component to request or trigger the execution of actions that require elevated privileges. For example, in an operating system, a user process that is not allowed to request the download of files from the Internet could trick the browser process into downloading a file on the process' behalf. In this case, the browser acts as the confused deputy, as it did not correctly check whether the requested action is permitted and performed actions on behalf of a user process that is not permitted to download files, allowing this process to bypass the operating system's security policy.

In general, a successful confused deputy attack requires the attacker to learn about the deputy's existence and capabilities and to find a way to make the deputy carry out actions on its behalf (actions that the attacker cannot perform themselves). Such requirements are easier to fulfill when one considers the public nature of the blockchain, which allows one to retrieve the bytecode of any deployed smart contract. These ideal conditions, in turn, enable the development of large-scale automated analyses that identify *confused contracts* and their associated possible targets.

The core intuition behind the *confused contract* vulnerability is as follows. First, the confused contract must include at least one call to another smart contract (as discussed in Section 2.4). Second, the attacker must be able to influence the arguments of such a call to control both the target contract and the target function. Third, the attacker-chosen function in the target contract must perform a security-sensitive action that depends on the identity of the caller (e.g., based on the value of `msg.sender`). As a result, the confused contract performs such actions on behalf of the attacker.

In this work, we focus on two scenarios: (1) a target contract performs persistent storage manipulations when called by a confused contract, or (2) assets of the confused contract stored in a target contract are directly transferred to an account of the attacker's.

The first scenario leverages the confused contract's privileges to modify the target contract's storage and can thus enable a more complex attack chain – for example, this happened in the PolyNetwork attack. On the other hand, the latter scenario has an immediate effect on the confused contract, whose assets are instantly lost.

3.1 Confused Contract Example

Consider a simple example showing how a developer mistakenly creates a smart contract vulnerable to a confused contract attack.

```
1  contract TradingBot
2  {
3   struct Op { address target; bytes calldata;}
4   function execute(Op[] memory ops) public
5   {
6     uint i;
7     for (i = 0; i < ops.length; i++)
8        ops[i][0].target.call(ops[i][1]);
9   }
10 }
```

Figure 2: `Solidity` code of the `TradingBot`. The `execute` function receives a list of tuples (Line 4) that are later used to make flexible, external function calls (in Line 8).

```
1
2  contract xyzToken
3  {
4   mapping (address => unit) public vault;
5
6   function transfer(address dst,uint256 val) public
7   {
8      address src = msg.sender;
9      require(vault[src] >= val);
10     vault[src] = sub(vault[src], val);
11     vault[dst] = add(vault[dst], val);
12     return true;
13  }
14
15  function balanceOf(address addr) public
16  {
17   return vault[addr]
18  }
19 }
```

Figure 3: `Solidity` code of a contract implementing a custom crypto-currency. The function `transfer`, provided by the standard ERC20 [42] interface, updates the persistent storage of the `xyzToken` contract according to the passed parameters. The function `balanceOf` returns the amount of token held by the input address *addr*.

Alice wishes to implement a trading bot by creating a contract called `TradingBot`. This contract maintains custody of some tokens and, when directed by the owner (who is Alice), it can exchange one type of token for another. Alice knows that the DeFi ecosystem is changing rapidly, and she would like to remain forward-compatible with future types of exchanges. However, the various exchanges all have different APIs, and moreover, she does not know what exchanges might exist in the future.

Alice decides to implement her trading bot as illustrated in Figure 2. Specifically, she writes an `execute` function that accepts a list of trading operations `ops` as input and then executes them in a loop. This design is actually directly inspired by similar bots on the Ethereum chain, and gives Alice the flexibility to construct the appropriate list of calls as she sees fit. In particular, Alice is free to encode any new exchange function's interface as she observes them deployed in the future.

Benign Use Case. In the regular use case, Alice uses her trading bot to exchange her tokens. For example, consider Figure 4, which shows Alice trying to exchange `xyz` tokens for `abc` tokens. The transaction begins with Alice invoking the `execute` function of the `TradingBot` with the appropriate arguments (`Tx1` in Figure 4). Instructed by Alice, the `TradingBot` directly invokes the `xyzToken`'s `transfer` function (Figure 3), which modifies the `xyzToken` contract's persistent storage to reflect that 100 `xyz` tokens are moved from the `TradingBot` account to the exchange account, `DEX`. This represents the payment for the token exchange (`iTx1` in Figure 4). After the first call, the `TradingBot` is instructed to call the `DEX` contract's `swap` function and specifies the amount of `abc` tokens it would like to receive as the result of the exchange operation (`iTx2` in Figure 4). `DEX` checks for the appropriate amount of payment (not illustrated here), and then transfers 50 `abc` tokens (represented by `iTx3` in Figure 4) to `TradingBot` using the `transfer` function in the `abcToken` contract, whose implementation is identical to the one in Figure 3. The transaction then completes.

Malicious Use Case. Due to the fact that the `TradingBot`'s `execute` function is marked as "public", and there is *no access control mechanism* in place, any unprivileged user can interact with it. Hence, an attacker, Mallory, can craft `CALLDATA` to invoke `TradingBot`'s function `execute` (`Tx2` in Figure 4). In her attack, Mallory instructs the `TradingBot` to call `xyzToken`'s function `transfer`, sending all of `TradingBot`'s `xyz` tokens balance to Mallory (`iTx4` in Figure 4). With the same process, Mallory can re-use the same attack to take ownership of all `TradingBot`'s `abc` tokens.

Discussion. The two fundamental enablers for the exploit of the confused contract vulnerability are the following: First, the identity of the `TradingBot` can be "stolen" to communicate with other contracts because the attacker controls the input to the call in the `execute` function (as illustrated in Figure 2, Line 5). Second, there exists on the blockchain a second contract (target contract) that holds assets on behalf of the `TradingBot`. Here, the `xyzToken` contract is an instance of a target contract. Both conditions are necessary for a confused contract vulnerability to exist.

For this specific example, to remove the vulnerability, Alice would need to validate the identity of any address that is interacting with her `TradingBot`. In particular, she could implement an access control routine that checks the value of `msg.sender` before performing any cross-contract interactions.

3.2 Confused Contract Attacks

In this work, we defined an attack as a *confused contract attack* if it satisfies the following three requirements:

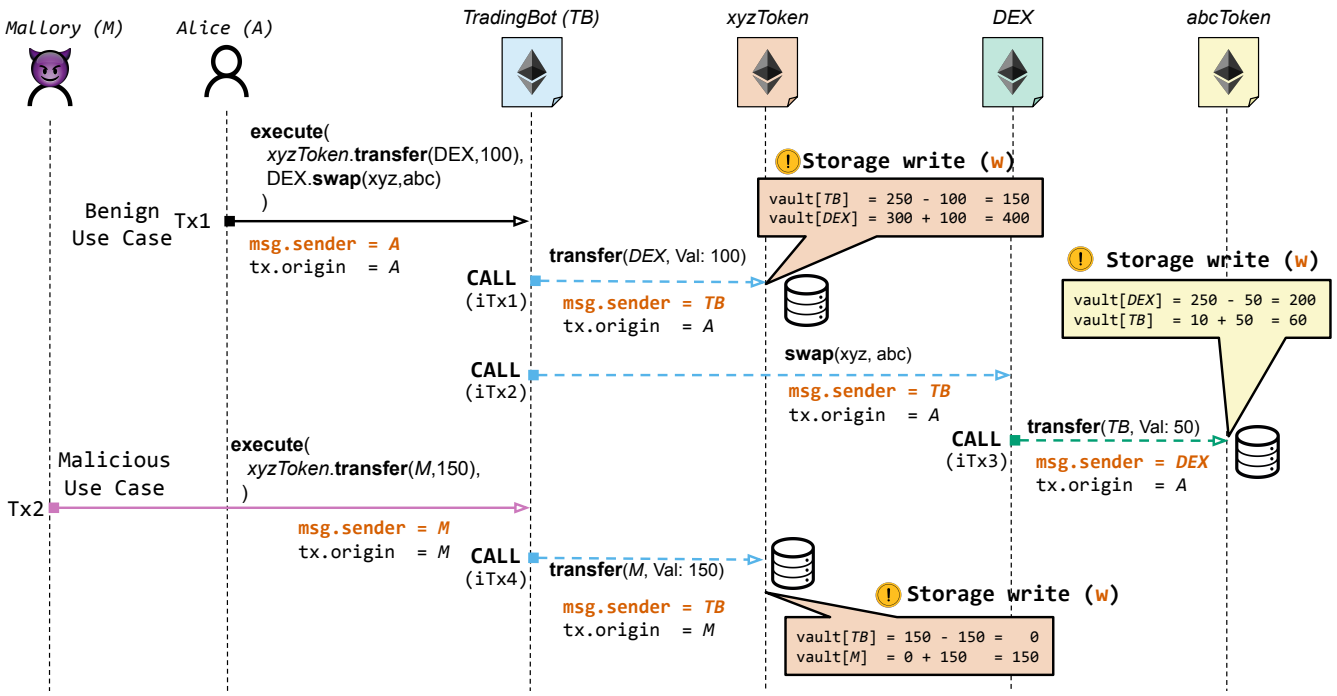

Figure 4: Overview of benign and malicious use cases when leveraging the `TradingBot` contract to operate on the blockchain. The solid arrows correspond to transaction `Tx`, while the dashed arrows correspond to internal transaction `iTx`. `TB` executes `iTx1-3` in response to `Tx1` sent by Alice `A`. Later, the attacker `M` sends a malicious transaction, `Tx2`, to `TB`, exploiting its identity to transfer all the `TB` tokens to `M`. In the malicious use case, the contract `TB` is the confused contract, and the contract `xyzToken` is the target contract.

(R1) A successful attack needs a pair of contracts: a confused contract C_c and a target contract C_t. The confused contract C_c contains a public function that serves as an entry point for the attacker.

(R2) The execution of a public function in C_c leads to a cross-contract invocation (a call) whose `tAddr` and `tFunc` arguments (see Section 2.4) can be controlled by the attacker.

(R3) The attacker uses their control over the "open" call in C_c to invoke a function in C_t, and this function performs modifications to the C_t persistent storage *only* when the value of `msg.sender` is the address of the confused contract C_c. That is, the target contract C_t associates some privileges (e.g., access to protected storage) with the confused contract C_c. By accessing the target contract *through* the confused contract, the attacker is able to trigger actions in C_t with the privileges of C_c.

When the three aforementioned requirements are satisfied, an attacker may perform security-critical actions, such as writing attacker-controlled values in the persistent storage of the target contract or directly drain the assets of the confused contract.

Given **(R3)**, to "steal" the identity of a confused contract during a cross-contract interaction, we want to consider only opcodes that update the `msg.sender` value at each internal transaction (`iTx`), *and* allow a C_t to perform modifications of its persistent storage. According to the semantics of the communication opcodes introduced in Section 2.4, `CALL` and `CALLCODE` are then the only opcodes that can be used for a confused deputy attack. In fact, a `STATICCALL` does not allow a C_t to perform write operations on the storage, while a `DELEGATECALL` does not update the `msg.sender` value when creating an `iTx`, and therefore we consider both of them out of scope. Furthermore, even if compliant with **R3**, we decided to ignore the `CALLCODE` opcode because it has been deprecated in the EVM in favor of `DELEGATECALL` [14]; in addition, it is used in less than 0.02% of the contracts in our dataset.

4 Approach

To identify the requirements necessary for a *confused contract* attack on the blockchain (**R1-3** discussed in Section 3.2), we developed KAI. Figure 5 presents an overview of our system. In the following paragraphs, we describe each one of the four steps in our analysis.

❶ **Bytecode Lifting and CFG Construction.** To begin, KAI takes the smart contract's EVM bytecode as input, lifts it to an intermediate registry-based representation, and runs a state-of-the-art CFG reconstruction analysis and constant propagation procedure using Gigahorse [27].

❷ **Call Inspector.** Once the CFG is built, KAI statically checks if the contract contains any CALL opcodes. If so, the system analyzes each one to understand if the CALL's arguments tAddr and tFunc can be controlled by the attacker. To do this, KAI first statically verifies that both arguments are *not* constant (if they are constant, they cannot be controlled by the attacker). Then, KAI uses the reconstructed CFG and callgraph to identify entry points (that is, public functions) that are connected to the CALL. Starting from an entry point, KAI initiates a symbolic execution directed toward the CALL opcode, using fully symbolic input. Note that the symbolic input represents the CALLDATA that an attacker would send as part of a (malicious) transaction Tx. After reaching the CALL opcode location, KAI extracts the path constraints (for the path from the entry point to the CALL). Using these constraints, the system asks the solver to find a solution for the symbolic CALLDATA that is used as the input. If such a solution exists, KAI attempts to infer the relationship between the CALLDATA and the tAddr/tFunc arguments of the CALL. Whenever both the tAddr *and* tFunc values can be chosen by the attacker, the two attack requirements (**R1** and **R2**) are satisfied. As a result, the contract under analysis is tagged as a confused contract candidate (C_c).

❸ **Path Feasibility Validator.** We then verify the feasibility of all confused contract warnings using the *Path Feasibility Validator*. This component verifies, using a local blockchain instance, if it is possible in practice to use the synthesized (concretized) CALLDATA to reach the target CALL. This allows the tool to remove warnings that stem from imprecisions in our symbolic execution.

❹ **Checkers.** Finally, we process the filtered warnings with two checkers to verify whether requirement **R3** holds: the *Generic Checker* and the *Token Checker*. The *Generic Checker* finds C_c-C_t pairs where the confused contract C_c has any special privilege to modify the target contract C_t's storage. Similarly, the *Token Checker* finds target contracts where the confused contract holds (or held in the past) any balance of digital assets. The warnings created by such checkers are then examined by an analyst to verify whether it is possible to create an end-to-end exploit.

In the following sections, we discuss our approach in more detail.

4.1 Call Inspector

The *Call Inspector* receives as input the intermediate registry-based representation of a contract's EVM bytecode and its CFG (from Step ❶). It locates and inspects all CALL operations and checks whether the contract meets the requirements **R1-2** for a confused contract vulnerability.

We first leverage the results of the constant propagation from Step ❶ (Figure 5) to check whether the tAddr/tFunc arguments are constants. In such cases, it would be impossible for an attacker to influence their values through the CALLDATA, hence we discard the CALL instance. Otherwise, we use symbolic execution to understand if an attacker's input can control the argument values.

The possible entry points for our analysis are all public functions of the smart contract for which a static path exists to the target CALL instruction. To identify such entry points, we use the previously-generated CFG and callgraph. After identifying a valid entry point (satisfying **R1**), we generate a fully-symbolic byte string of 1,024 bytes and set it as the input argument of the corresponding function. Note that the generated symbolic byte string models the CALLDATA that an attacker would send to the smart contract in a transaction.

At this point, we begin the symbolic exploration of the smart contract's code from the selected entry point.

4.1.1 Symbolic Exploration

The goal of the symbolic execution analysis is to determine whether an attacker can control the values of tAddr and tFunc for one of the CALL instructions (this is **R2**). To this end, we designed a symbolic execution engine augmented with a set of features that allow us to fine-tune the code exploration, aiming at a sweet spot between generality (finding as many confused contracts as possible) and performance (avoiding path explosion). Our engine employs a state-of-the-art, fully symbolic memory model [53] with an extension proposed by Falke et al. [22]. In the following, we introduce a few important features (**F**) of our symbolic execution engine.

(F1) Directed Exploration. We employ directed symbolic execution [40] toward previously-identified target CALL instructions, using the inter-procedural CFG provided by ❶. In particular, during the symbolic exploration, we use static information from the CFG to prune all paths that do not lead to the target CALL opcode.

(F2) Partially Concrete Storage. The execution of a smart contract's code does not happen in a vacuum. That is, in addition to the input provided in the transaction (as discussed in Section 2.4), the execution also depends on the state of the blockchain.

We extend our symbolic storage model with *partially concrete storage* support. This means that when the program reads from storage at a *concrete* index, we fetch the corresponding concrete value(s) from the blockchain at a certain (fixed) block number[1]. Such a storage model allows us to proactively discard states that are not reachable given the on-chain storage values.

(F3) Partially Concrete Execution Context. This feature is related to the previous one (**F2**). Specifically, we keep the input to a function (its CALLDATA) symbolic. However, our execution engine allows for the partial concretization of the execution context (in addition to the storage). That is, we set the msg.sender and the tx.origin to a concrete

[1]To facilitate reproducibility, we fix an arbitrary reference block number 16380000. We discuss this choice in Section 6.

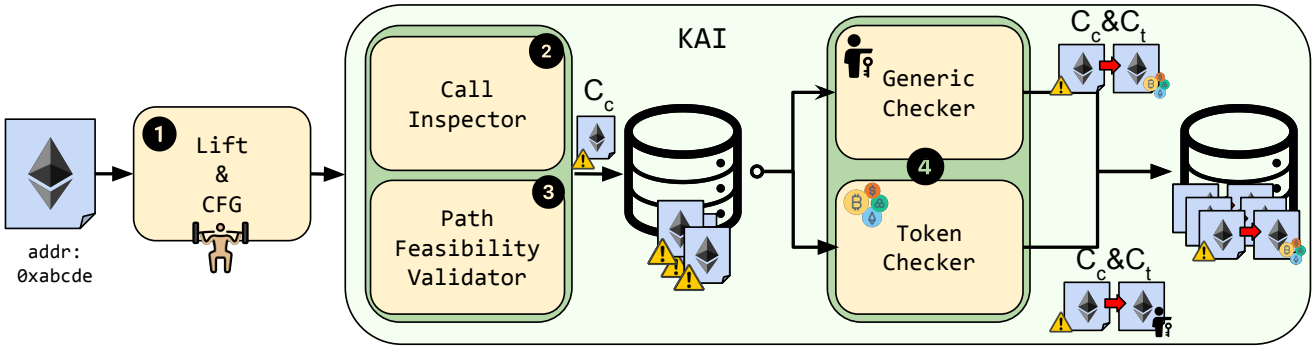

Figure 5: KAI Overview. The analysis pipeline follows the order of the circled numbers. Components ❶-❸ verify the requirements **R1-2** for a confused contract vulnerability. Component ❹ verifies instead **R3**.

address value that we control. Again, this allows us to discard symbolic states that are not reachable given the on-chain state.

(F4) Precise Handling of SHA3. The SHA3 hash function is frequently used in smart contracts. For example, the tFunc argument of a CALL can be computed by taking the SHA3 of a string (the function name), and keeping the first 4 bytes of its result (as discussed in Section 2.4). In fact, SHA3 is used so frequently that the EVM includes a dedicated opcode [65]. Unfortunately, the precise handling of the SHA3 opcode poses a non-trivial challenge for symbolic execution. Given an offset in memory and a size S, the SHA3 opcode calculates the KECCAK256 hash of the S bytes starting at the target offset. This cryptographic operation cannot be symbolically analyzed [11]. For this reason, different strategies were proposed to address this challenge [5, 25, 32, 38, 47]. In our symbolic execution engine, we use the approach proposed by Frank et al. [25]. Whenever we obtain solutions for the CALLDATA input, we also attempt to get solutions for the SHA3 operations observed during the symbolic execution. We do this by first concretizing the size S and the corresponding input buffer at the target offset and then calculating the value of the KECCAK256 operation, which is assigned to the result variable of the SHA3 opcode.

4.1.2 Constraint Solving

When our symbolic execution engine reaches the target CALL operation, we inspect the symbolic state at this point. Specifically, we extract the path constraints and query our underlying solver, Yices2 [13], to obtain a concrete solution (concrete values) for the symbolic CALLDATA, as well as for the tAddr and tFunc arguments.

If the solver is not able to find a solution, this means that we cannot provide an input (CALLDATA) that reaches the CALL. On the other hand, if the solver can determine a solution, we have found an input that reaches the CALL, together with concrete values for the CALL's arguments.

To understand if the argument values of tAddr *and* tFunc directly depend on the CALLDATA, and hence, are under the attacker's control, we employ a strategy based on finding path-preserving CALLDATA modifications. To this end, we first check whether the tAddr/tFunc arguments can be influenced by any symbolic values that do *not* come from CALLDATA. Such values can be return values from external contract calls (that are not modeled) or storage reads with a symbolic index. To identify these cases, we force the solver to fix the initial concrete solution for CALLDATA while generating a *different* solution for tAddr and tFunc. If the solver can change the values of tAddr and tFunc without changing CALLDATA, we assume that something else along the path influences such arguments. Otherwise, we assume that the values of tAddr and tFunc only depend on the CALLDATA.

However, it is not enough to prove that the tAddr and tFunc arguments depend on the CALLDATA. In addition, we also require that the attacker has some freedom over the actual values of the arguments. To check this, we query the solver for a *new and different* solution for the symbolic variables in CALLDATA, tAddr, and tFunc. If the solver can find a satisfying assignment, we conclude that there is a way for an attacker to influence the values of tAddr and tFunc via bytes in the CALLDATA. This indicates that **R2** is satisfied.

The astute reader will observe that we only checked for two different solutions. In theory, this is a very narrow definition of "sufficient freedom" over all possible argument values. However, we find that, in practice, most contracts either enforce one specific value or provide complete freedom for the attacker. Hence, checking for two solutions serves as a good proxy for complete freedom.

4.2 Path Feasibility Validator

Our symbolic execution analysis is not sound. For example, we do not follow calls to external functions (in other contracts), and we use partially concrete storage and execution context. Hence, in some cases, KAI reports that an attacker can reach and control a target CALL instruction when this is not possible. This can occur when, for instance, the confused contract's code invokes an *external* contract to sanitize the

value of the msg.sender before reaching the target CALL. That is, the value of the msg.sender is sent to an external "governance contract" that checks if the account is allowed to operate on the confused contract. Of course, if the smart contract included the check directly, we would properly detect this case.

To filter out false positives, such as the "governance contract" cases above, we leverage an implementation of the EVM [21] to execute the confused contract's bytecode, supplying the concrete CALLDATA value generated during *Call Inspection* (Figure 5, ❷) as input. In particular, the validator provides a complete execution environment to the contract, simulating its execution on the blockchain and accounting for checks performed outside of its code. If this step is successful, we have strong evidence that the synthesized CALLDATA allows us to reach the target CALL instruction. We provide a more complete discussion of potential false negatives and false positives in Section 5.

4.3 Checkers

At this point, we have identified a confused contract that allows an attacker to control the arguments of a call invoking an external function. While this is of concern, one more factor is still required for a successful attack. Such a factor is a second (target) contract that holds some privilege (e.g., access, assets) on behalf of the confused contract—a requirement that is captured by **R3**. To this end, we apply two analyses: a *Generic Checker* and a *Token Checker*.

Generic Checker. The purpose of this checker is to identify cases where a target contract has allocated some state on behalf of the confused contract.

Finding such target contracts C_t on the blockchain is not a trivial task. In fact, one would potentially have to analyze *all* existing contracts (∼54 million contracts at the time of writing) to determine whether they have any relationship with a given confused contract C_c. To simplify this process and make it scale, we narrow our search to only those contracts that had any prior interactions with C_c. More precisely, we scan the blockchain history and extract all the addresses of contracts C_t that were the target of an internal transaction (iTx) coming from a given C_c. We believe that this makes intuitive sense: If a certain target contract allocates state on behalf of another source contract (or user account), such source contract (or user) will likely have invoked the target contract.

The *Generic Checker* first replays each historic transaction iTx – which was sent by the confused contract – in a locally simulated blockchain environment and collects an execution trace. Then, it replays the transaction again, but this time, we change the msg.sender of the iTx to an arbitrary address. If we detect any differences between the two executions in terms of writes to persistent storage, we raise a warning.

Token Checker. The goal of the *Token Checker* is to determine whether the confused contract holds some tokens (cryptocurrency) that an attacker might be able to steal. In contrast to the *Generic Checker*, the *Token Checker* can detect relationships between a confused contract C_c and a target contract C_t even if they *never* interacted. This is because we can limit our analysis to target contracts with certain types of digital assets. To make this analysis scale, we limit the analysis to token contracts based on the ERC-20 and ERC-721 standards [42, 44]. These standards define interfaces used by developers to write smart contracts that implement custom digital assets living on the blockchain (e.g., cryptocurrency tokens, NFTs). The goal of the *Token Checker* is to understand if a given confused contract C_c currently holds (or held in the past) any digital assets. When this is true, we raise a warning.

A straightforward way to identify the number of tokens owned by a confused contract C_c is to call the standardized method balanceOf available in ERC-20/ERC-721-based contracts. However, this approach would be very inefficient. In fact, one would have to call the balanceOf methods of all possible token contracts C_t, for every possible block. Instead, we collect the balance information using the *transfer event logs*, a standardized log event emitted on the blockchain whenever an ERC-20/ERC-721 token is transferred to, or from, a contract. Summaries of such log events allow us to determine the number of tokens that were owned by a confused contract at any point in time.

5 Evaluation

For all our experiments, we use three servers equipped with 300Gb of RAM and dual Intel(R) Xeon(R) Gold 6330 CPUs. We use GNU Parallel [56] to parallelize our tasks.

Dataset Information. We extracted all smart contract addresses created in the 24-month period from December 2020 to December 2022[2]. This accounts for a total of 2,335,193 smart contracts. When trying to obtain their bytecode, we observed that 86,525 contracts (∼3.84%) self-destructed in the same block when they were created. Thus, we were left with a total of 2,248,668 smart contract binaries. Within our dataset, we identified 307,957 (∼13.69%) contracts with *unique* SHA256 hashes of their respective bytecode.

To motivate our decision to perform binary-only analysis (instead of relying on source code), we collected information regarding the availability of source code for the contracts in our dataset. We observe that when considering only unique values for the contracts' bytecode, source code is not available for 35.8% of them. Interestingly, when looking at all contracts (without discarding duplicates), the percentage of contracts with no source code increases to 48.4%. In Figure 6, we show the availability of source code as it relates to the size of the contracts' bytecode. This data suggest that the fraction of contracts without source code decreases as the contracts grow

[2]Corresponding to blocks in the range 11363270–16086235.

larger. Nevertheless, even with large dApps, source code is frequently not available, which supports our decision to implement binary-only program analysis techniques. In Table 1, we show a summary of source code availability with a split at 5Kb for the size of contracts' bytecode's.

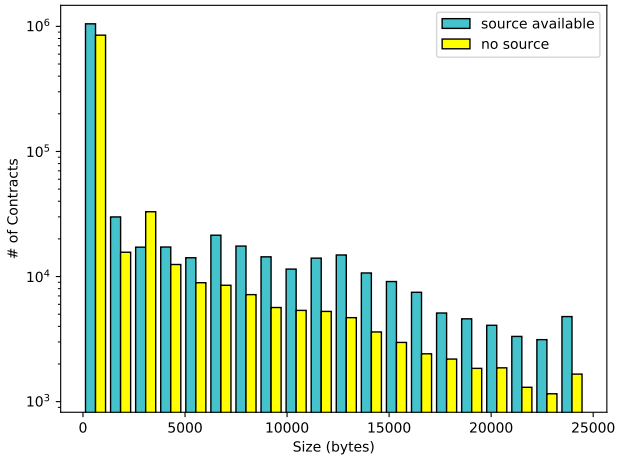

Figure 6: Correlation between bytecode size and availability of source code for non-unique contracts.

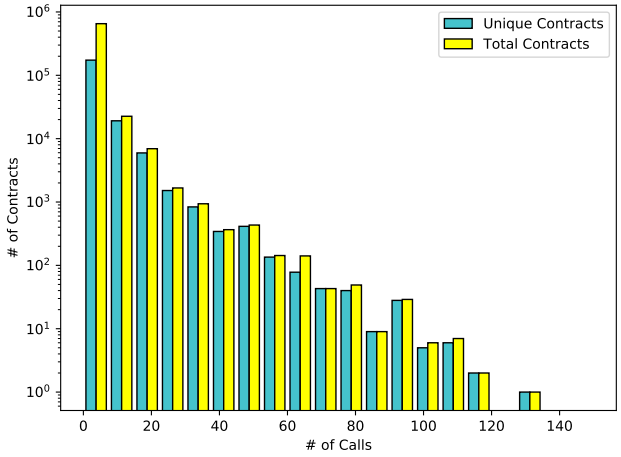

Figure 7: Distribution of the number of CALL opcodes for unique and non-unique contracts in our dataset.

5.1 Confused Contracts

Static Analysis. We obtained CFG reconstruction and static analysis results by running Gigahorse [27] on all the contracts with a timeout of 30 minutes and a memory limit of 50 Gb. After this analysis, we discarded 17,996 (0.77%) contracts for which Gigahorse failed to provide results and remove an additional 1,538,365 (65%) smart contracts that do not contain *any* CALL opcode. As illustrated by Figure 7, many of the remaining contracts include only a relatively small number of CALL instructions, but with some outliers that contain a considerable amount. We advance to the next analysis step a total of 692,307 smart contracts.

	Has Source		No Source	
	Unique	Total	Unique	Total
Small Size (<5Kb)	58,869	1,113,287	54,933	912,392
Large Size (≥5Kb)	138,765	159,070	55,390	63,919

Table 1: Source code availability for smart contracts in our dataset considering unique and non-unique code.

Call Inspector. We ran our *Call Inspector* analysis (discussed in Section 4.1) on all 692,307 contracts. Given a contract, we identify all its CALL opcodes and discard the ones for which Gigahorse reported that *at least one* of the tAddr or tFunc arguments is constant. For the remaining calls, we performed symbolic execution using a timeout of 30 minutes. As we discuss in Section 4.1.1 (**F2&F3**), our analysis extracts concrete values from the persistent storage whenever we access it with a fully concrete index. Since concrete persistent storage values can be different for contracts even when their bytecode is identical, for this analysis stage we consider non-unique contracts.

Results Discussion. The (symbolic execution) analysis of all contracts took approximately 54 hours, and it processed a total of 780,836 CALL(s)[3]. As illustrated in Figure 8, most of the CALLs were analyzed in less than a minute, while our engine encountered a timeout for 29,400 CALL instances (3.77%). KAI flagged 416,163 CALLs (53.3%) as unreachable during symbolic execution (e.g., there are checks on msg.sender preventing arbitrary accounts from reaching the call), 333,633 CALLs (42.73%) reachable with non-controllable arguments of the CALL, and, finally, 1,640 (0.21%) reachable CALL opcodes with controllable tAddr *and* tFunc. Such CALLs belong to contracts that are potentially vulnerable to a confused contract attack. To shed some light on the potential misclassification of controllable CALL opcodes as non-controllable, we randomly sample 20 CALLs in distinct contracts and manually verify whether they are indeed non-controllable. For all samples, we confirm the results of our analysis.

In the next step, we used the *Path Feasibility Validator* on the 1,640 contracts. This step reduced the number to 529 confused contracts for which we can replay inputs that reach the CALL instruction (and the analysis believes that the call arguments are under the attacker's control). As discussed in Section 4.2, our validation checks the reachability of a target CALL opcode, but it does not consider whether a transaction successfully terminates – without reverting – after having reached the call. Hence, false positives are still possible, e.g., the msg.sender might be checked *after* the call.

[3] A contract can contain multiple CALL opcodes.

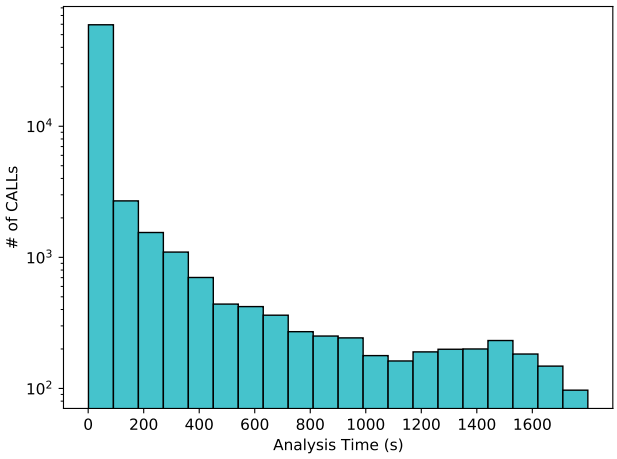

Figure 8: Distribution of the analysis time spent by symbolic execution to mark a `CALL` as controllable or non-controllable.

5.2 Checkers

In the final step, we applied our two checkers (as described in Section 4.3) to the 529 confused contracts.

Generic Checker. The *Generic Checker* identified 32 contracts with *at least one* historic interaction on the blockchain in which differences in persistent storage writes were observed when changing the `msg.sender`. Specifically, we detected some potentially-sensitive persistent storage writes which could be performed by the confused contract but not by an arbitrary contract.

Token Checker. The *Token Checker* reported that 52 contracts held a certain amount of cryptocurrency tokens (ERC20 or ERC721 contracts) for *at least one block*. In this scenario, a successful exploitation of a confused contract attack can, or could have, directly caused the transfer of digital assets from the confused contract to the attacker's account.

5.3 Confused Contract Exploitation

In Section 5.2, we reported that KAI found 84 confused contracts and associated target contracts. However, even if our tool successfully reported conditions for confused contract attacks, one important question remains: *is it possible in practice to generate an exploit?* To answer this question, we manually investigated the flagged confused contracts and attempted to write end-to-end exploits. One peculiarity of smart contracts' exploitation is that once the effects of the exploit are observed, i.e., modifications to the persistent storage, they will be committed only if the continuation of the execution terminates successfully (i.e., it does not revert). Hence, even if it is sometimes possible to obtain favorable conditions for an exploit to produce its effects, it might not be feasible to reach the end of the execution without reverting its changes, which prevents any possible harm. Therefore, we confirmed all exploits created at this step by executing them in a local version of a real blockchain environment [21].

Generic Checker Warnings. After looking into the 32 warnings for confused contracts generated by the *Generic Checker*, we found that one of them interacted in the past with the Seaport [36] smart contract: an ERC-721 token marketplace. We crafted an exploit that forces the confused contract to increment an order counter (associated with the confused contract) in the persistent storage of the Seaport contract. It seems that this could lead to security issues. Specifically, in Figure 9, we show a snippet from the Seaport contract, where a developer left a comment stating that the order increment operation is sensitive and restricted to offerers, as it leads to a cancellation of their orders. Note that the vulnerability *is not* to be attributed to the Seaport Marketplace, but, rather, to the confused contract with controllable `CALL`'s arguments.

```
1  /*  @notice
2   *  Cancel all orders from a given
3   *  offerer with a given zone in bulk
4   *  by incrementing a counter.
5   *
6   *  Note that only the offerer may increment the
        counter.
7   *
8   * @return newCounter The new counter.
9   */
10 function incrementCounter() external returns
              (uint256 newCounter);
```

Figure 9: The `incrementCounter` function in Seaport. The developers explicitly noted that *only an offerer may increment the counter*. A confused contract can be forced by any unprivileged user to violate this requirement.

For the remaining instances in this category, we could not create exploits that would cause any harm. In 28 cases, the confused contract did check the `msg.sender` *after* the controllable `CALL` instruction but before terminating its execution (and, thus, the contract reverts). In 2 other cases, we could not fully control the target `CALL` instruction because of the additional checks on the `CALL` arguments that were not captured during our symbolic analysis. For example, in some cases, only specific solutions were allowed to be used as the `tAddr` in the confused contract call, preventing us from calling the identified target contracts and their functions. Finally, we found one case where the attacker can control the `CALL`, but we could not find any security-relevant writes to persistent storage in the target contracts. However, in this case, the confused contract is vulnerable and could be exploited in the future.

Tokens Checker Warnings. The *Token Checker* identified 52 confused contracts. We first ordered these contracts by the maximum total value in USD dollars that they held at *any point in time*. We then manually examined all contracts with

a value greater than $100, which yielded 23 contracts. During this analysis, we successfully developed exploits for 12 of them. That is, we were able to demonstrate the possibility of transferring the assets owned by a confused contract (for a specific token) to an account of an attacker's choice. All our exploits work by manipulating the input of the confused contract's public function so that, in the arguments of the CALL, the target address `targetAddress` is the address of a token contract and the target function `tFunc` is the `transfer` function of the contract [43], which sends the assets of the confused contract to the attacker-specified address.

In Table 2, we show the results of our findings. Note that we report the maximum values of US dollars associated with the tokens held by a confused contract, but we detected many other exploitation opportunities for smaller values and different time windows. Moreover, the identified exploits also allow the attacker to compromise any assets the confused contracts could have in the future. Regarding the 11 warnings for which we could not develop an exploit, 3 of them managed a total value of approximately a million dollars. However, we were unable at the time of writing to create successful attacks due to the high complexity of their logic. For 8 of them, we were unable to set the `tFunc` to the right contract target. Interestingly, all the confused contracts that we could exploit do not have source code available. This strengthens our belief that we need to develop analysis techniques that directly operate on EVM bytecode to improve the security of the Ethereum ecosystem.

#	Token	Block	Span	Val
1	DAI [12]	14104828	146	$838,436
2	DAI [12]	11469710	476	$190,090
3	BAS [2]	11454250	2	$12,610
4	Hegic [29]	11492550	42	$10,007
5	wETH [63]	13574103	12,199	$2,404
6	wETH [63]	15625577	14,867	$1,438
7	wETH [63]	15645205	651	$1,405
8	wBTC [62]	15200611	1	$993
9	LooksRare [35]	14634155	1,927,641	$883
10	LUSD [37]	15451410	62	$391
11	wETH [63]	15596427	1146	$261
12	wETH [63]	14170312	19,396	$133
Total				**$1,058,961**

Table 2: Summary of the discovered exploits for the confused contract attacks. Token represents the token held by the contract starting from the block in Block for Span blocks. Val is an estimate in USD dollars of the value of the tokens at the specific Block in which the opportunity first appeared. We fetched the price per token using the Uniswap [60] contract.

Ethical Concerns. As some of our exploits can still compromise assets on the blockchain, we took some precautions. First, when developing our exploits, we *never* execute them in any public blockchain. Rather, all the exploits were tested in a local environment that models the real one. Second, when reporting our results in Table 2, we redacted the confused contracts' addresses. Finally, in the spirit of responsible disclosure, we attempted to contact the entities behind exploitable confused contracts—whenever possible—and are currently awaiting acknowledgment. However, as the individuals behind a smart contract are often unknown, it is not always practical to report a vulnerability.

6 Discussion and Limitations

Directed Exploration. The correctness of our directed exploration—discussed in Section 4.1.1 (**F1**)—relies on the precision of the underlying state-of-the-art CFG reconstruction framework [27]. While it is true that inconsistencies in the CFG analysis would lead to both false positives and false negatives in our results, we did not observe such a problem in practice.

EntryPoint Selection. While multiple entry points for each target CALL can exist, our symbolic analysis selects only one. KAI could be extended fairly straightforwardly to take into account more possible paths, which might yield additional confused contracts. However, in practice, we found that many contracts only have a few relevant entry points for each CALL.

Reference Block Choice. As described in Section 4.1.1 (**F2**), we use an arbitrary reference block number 16380000 throughout our analyses. While running our analyses on each block would be ideal, it is impractical to do so. Moreover, the benefits of fixing an arbitrary reference block are twofold. First, this allows for the reproducibility of our results. Second, choosing such a recent block allows us to verify that the vulnerability is currently exploitable.

Constraint Analysis. Our path-preserving-transformation heuristic (Section 4.1.2) marks `tAddr` and `tFunc` as controllable even if the attacker cannot manipulate their values arbitrarily. Instead, our heuristic is used to provide evidence of a connection between values in CALLDATA supplied by the attacker and the values of the CALL arguments, which corresponds to our attacker model described in Section 3.2.

Checker Warnings. Our checkers produced a total of 84 warnings for possible confused contracts. However, this number can be increased by allowing KAI to further explore the attack surface of confused contract vulnerability via additional or extended checkers.

Full Exploit Synthesis. While our system automatically provides the CALLDATA to reach a controllable CALL in a confused contract, the creation of an end-to-end exploit for a confused deputy attack is currently a manual task.

7 Related Work

In this section, we review related work focused on finding vulnerabilities in smart contracts.

7.1 Individual Contract Analysis

Several systems have been proposed – both by the academic research community and industry – to identify vulnerabilities in the code of individual smart contracts.

Static Analysis. Grech et al. proposed Gigahorse [27], a static analysis framework that translates the stack-based bytecode of a smart contract into a register-based intermediate representation that can be used for decompilation. Gigahorse provides precise CFG reconstruction and numerous out-of-the-box data-flow analyses that have been used to build effective bug-hunting systems in the past [4, 26, 33, 54]. A similar approach has been proposed by Fesit et al. with Slither [23], a static analysis framework that targets `Solidity` [55] source code and supports the detection of a multitude of different bug classes. Tikhomirov et al. proposed SmartCheck [58], a system that translates `Solidity` source code into an XML representation and identifies bugs with the help of xPath patterns. Finally, Tsankov et al. proposed Securify [59], a framework that leverages a smart contract's dependency graph to identify patterns of vulnerabilities.

Symbolic Execution & Bounded Model Checking. The first effort in this area has been proposed by Luu et al. with Oyente [38], followed by Krupp et al. [32] with teEther. In particular, Oyente aimed at finding bugs such as reentrancy and transaction-ordering dependence. teEther targeted a similar class of bugs, but its symbolic execution engine is equipped with an automatic exploit generation system. In addition, industry solutions such as Mytrhil [10] and Manticore [41] have been proposed and became the de-facto standard tools in the arsenal of professional smart contracts auditors. Finally, Bose et al. proposed Sailfish [3], a symbolic execution system for automatically finding state-inconsistency bugs.

Fuzzing. Many systems based on fuzzing techniques have been developed through the years with the majority of them requiring `Solidity` source code: Echidna [28], Harvey [66], ChainFuzz [7], Foundry [24], and fuzzing-like-a-degen [1]. Currently, we are aware of only one black-box fuzzer: ContractFuzzer [31].

While KAI leverages ideas from prior static analysis and symbolic execution systems (that inspect individual contracts), our goal is different. We aim to detect inter-contract bugs that belong to a new class of bugs that we call *confused contract vulnerabilities*.

7.2 Inter-Contract Analysis

While the vast majority of prior work focuses on the detection of intra-contract bugs, some work has been done to expand the analysis to inter-contract vulnerabilities.

Ma et al. proposed Pluto [39], a system that first reconstructs the inter-contract CFG of smart contracts that interact with each other. Pluto then leverages a symbolic analysis that explores the CFG to identify different kinds of classic bugs, e.g., reentrancy, and arithmetic issues. In a similar vein, Frank et al. proposed ETHBMC [25], a bounded model checker that supports the symbolic execution of a smart contract bytecode even when external interactions are present. Liao et al. proposed SmartDagger [34], a static analysis framework that combines different data-flow analyses and target functions' optimizations to discover inter-contract vulnerabilities. Similarly, Ye et al. proposed Clairvoyance [67], a static analysis tool based on source code that constructs a cross-contract CFG and then automatically identifies candidate critical paths for the exploitation of reentrancy bugs. Finally, Xue et al. proposed xFuzz [68], a system that uses a machine-learning model to filter out benign cross-contracts execution paths to eventually increase the efficiency of cross-contract fuzzing.

Different from all this work, our approach is not focused on the analysis of the smart contract's code to identify classic vulnerabilities like reentrancy, but rather, KAI targets a logic cross-contract vulnerability that arises when particular requirements are met.

8 Conclusions

In this paper, we introduced the *confused contract* vulnerability class, a variant of the confused deputy bug performed in the context of smart contracts running on the blockchain. In particular, we show how contracts that contain controllable invocations of other contracts' functions (i.e., confused contracts) can be exploited by unprivileged users to perform unwanted actions on their behalf, e.g., financial assets transfers from the confused contract to an attacker.

To identify *confused contract* attack opportunities in the wild, we developed KAI, a system that we used to analyze 2,335,193 million smart contracts created on the Ethereum blockchain between block 11363270 and block 16086235. KAI raised a warning for a total of 529 smart contracts, flagging them as possibly vulnerable to a confused contract attack.

To identify exploitation opportunities for confused contract attacks in the blockchain, we developed two checkers that helped us to automatically identify target contract candidates. Based on these findings, we were able to develop 13 past and present working exploits for confused contract attacks in the real world, revealing the potential of compromising digital assets for more than a million US dollars. With this work, we wanted to present a few use cases for exploiting a *confused contract* vulnerability. However, we believe that the scale of the problem is far-reaching and that improved tools are necessary to comprehend the intricate relationships present within the numerous multi-contract interactions occurring daily on the blockchain.

References

[1] 0xalpharush. Barebones solidity smart contract fuzzer. https://github.com/0xalpharush/fuzzing-like-a-degen, 2022.

[2] BAS. Bas. https://etherscan.io/token/0xa7ED29B253D8B4E3109ce07c80fc570f81B63696, 2023.

[3] 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. IEEE, 2022.

[4] Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. Ethainter: a smart contract security analyzer for composite vulnerabilities. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 454–469, 2020.

[5] 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. *arXiv preprint arXiv:1809.03981*, 2018.

[6] Chainlink. Reentrancy attacks and the dao hack. https://blog.chain.link/reentrancy-attacks-and-the-dao-hack/, 2022.

[7] ChainSecurity. Chainfuzz: fast transaction fuzzer for ethereum smart contracts. https://github.com/ChainSecurity/ChainFuzz, 2019.

[8] Deric Cheng. What is gas and how is it used? https://www.web3.university/tracks/create-a-smart-contract/what-is-gas-and-how-is-it-used, 2023.

[9] Cointelegraph. Immunefi says it has facilitated $66m in bug bounties since inception. https://cointelegraph.com/news/immunefi-says-it-has-facilitated-66m-in-bug-bounty-payouts-to-whitehats-since-inception, 2022.

[10] ConsenSys. Mythril. https://github.com/ConsenSys/mythril, 2022.

[11] Ricardo Corin and Felipe Andrés Manzano. Efficient symbolic execution for analysing cryptographic protocol implementations. In *International Symposium on Engineering Secure Software and Systems*, pages 58–72. Springer, 2011.

[12] DAI. Dai. https://etherscan.io/token/0x6b175474e89094c44da98b954eedeac495271d0f, 2023.

[13] Bruno Dutertre. Yices 2.2. In *International Conference on Computer Aided Verification*, pages 737–744. Springer, 2014.

[14] Ethereum. Eip-7: Delegatecall. https://eips.ethereum.org/EIPS/eip-7, 2015.

[15] Ethereum. Block and transaction properties. https://docs.soliditylang.org/en/develop/units-and-global-variables.html#block-and-transaction-properties, 2022.

[16] Ethereum. Decentralized finance (defi). https://ethereum.org/en/defi/, 2022.

[17] Ethereum. Ethereum. https://ethereum.org/en/, 2022.

[18] Ethereum. What is eth? https://ethereum.org/en/eth/, 2022.

[19] Ethereum. Ethereum accounts. https://ethereum.org/en/developers/docs/accounts/, 2023.

[20] Ethereum. Gas and fees. https://ethereum.org/en/developers/docs/gas/, 2023.

[21] Ethereum. A python implementation of the ethereum virtual machine. https://github.com/ethereum/py-evm, 2023.

[22] Stephan Falke, Florian Merz, and Carsten Sinz. Extending the theory of arrays: memset, memcpy, and beyond. In *Working Conference on Verified Software: Theories, Tools, and Experiments*, pages 108–128. Springer, 2014.

[23] 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, 2019.

[24] Foundry. Fuzz testing. https://book.getfoundry.sh/forge/fuzz-testing?highlight=fuzzing#fuzz-testing, 2023.

[25] Joel Frank, Cornelius Aschermann, and Thorsten Holz. ETHBMC: A bounded model checker for smart contracts. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2757–2774, 2020.

[26] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Madmax: Analyzing the out-of-gas world of smart contracts. *Communications of the ACM*, 63(10):87–95, 2020.

[27] Neville Grech, Sifis Lagouvardos, Ilias Tsatiris, and Yannis Smaragdakis. Elipmoc: advanced decompilation of ethereum smart contracts. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA):1–27, 2022.

[28] Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. Echidna: effective, usable, and fast fuzzing for smart contracts. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 557–560, 2020.

[29] HEGIC. Hegic. https://etherscan.io/address/0x584bC13c7D411c00c01A62e8019472dE68768430, 2023.

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

[31] Bo Jiang, Ye Liu, and Wing Kwong Chan. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 259–269. IEEE, 2018.

[32] Johannes Krupp and Christian Rossow. teEther: Gnawing at ethereum to automatically exploit smart contracts. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1317–1333, 2018.

[33] Sifis Lagouvardos, Neville Grech, Ilias Tsatiris, and Yannis Smaragdakis. Precise static modeling of ethereum "memory". *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–26, 2020.

[34] 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, 2022.

[35] LOOKSRARE. Looksrare. https://etherscan.io/token/0xf4d2888d29d722226fafa5d9b24f9164c092421e, 2023.

[36] luckytrader. What is seaport? opensea's new nft marketplace protocol. https://etherscan.io/address/0x00000000006c3852cbEf3e08E8dF289169EdE581#code, 2023.

[37] LUSD. Lusd. https://etherscan.io/token/0x5f98805A4E8be255a32880FDeC7F6728C6568bA0, 2023.

[38] 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, 2016.

[39] 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*, 2021.

[40] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S Foster, and Michael Hicks. Directed symbolic execution. In *International Static Analysis Symposium*, pages 95–111. Springer, 2011.

[41] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1186–1189. IEEE, 2019.

[42] OpenZeppelin. Erc20. https://docs.openzeppelin.com/contracts/3.x/erc20, 2022.

[43] OpenZeppelin. Erc20. https://docs.openzeppelin.com/contracts/3.x/api/token/erc20#IERC20-transfer-address-uint256-, 2022.

[44] OpenZeppelin. Erc721. https://docs.openzeppelin.com/contracts/3.x/erc721, 2022.

[45] OpenZeppelin. Reentrancyguard. https://docs.openzeppelin.com/contracts/4.x/api/security#ReentrancyGuard, 2022.

[46] OpenZeppelin. Safemath. https://docs.openzeppelin.com/contracts/2.x/api/math, 2022.

[47] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. Verx: Safety verification of smart contracts. In *2020 IEEE symposium on security and privacy (SP)*, pages 1661–1677. IEEE, 2020.

[48] PolyNetwork. Enhancing connections between ledgers by providing interoperability in web 3.0. https://poly.network/, 2023.

[49] Kaihua Qin, Liyi Zhou, Yaroslav Afonin, Ludovico Lazzaretti, and Arthur Gervais. Cefi vs. defi–comparing centralized to decentralized finance. *arXiv preprint arXiv:2106.08157*, 2021.

[50] Rekt. Rekt, leaderboard. https://rekt.news/leaderboard/, 2022.

[51] Kudelski Security Research. The poly network attack explained. https://research.kudelskisecurity.com/2021/08/12/the-poly-network-hack-explained/, 2021.

[52] Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. ethor: Practical and provably sound static analysis of ethereum smart contracts. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 621–640, 2020.

[53] Carsten Sinz, Stephan Falke, and Florian Merz. A precise memory model for Low-Level bounded model checking. In *5th International Workshop on Systems Software Verification (SSV 10)*, 2010.

[54] Yannis Smaragdakis, Neville Grech, Sifis Lagouvardos, Konstantinos Triantafyllou, and Ilias Tsatiris. Symbolic value-flow static analysis: deep, precise, complete modeling of ethereum smart contracts. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–30, 2021.

[55] Solidity. Solidity. https://docs.soliditylang.org/, 2023.

[56] O. Tange. Gnu parallel - the command-line power tool. *;login: The USENIX Magazine*, 36(1):42–47, Feb 2011.

[57] theverge. Crypto collapse: Ftx's fall is one piece of a long, cold, contagious crypto winter. https://www.theverge.com/2022/11/10/23450169/crypto-winter-ftx-binance-celsius-bitcoin, 2022.

[58] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. Smartcheck: Static analysis of ethereum smart contracts. In *Proceedings of the 1st international workshop on emerging trends in software engineering for blockchain*, pages 9–16, 2018.

[59] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 67–82, 2018.

[60] Uniswap. Uniswap. https://uniswap.org/, 2023.

[61] Vyper. Vyper. https://vyper.readthedocs.io/en/stable/, 2023.

[62] WBTC. Wbtc. https://etherscan.io/address/0x2260FAC5E5542a773Aa44fBCfeDf7C193bc2C599, 2023.

[63] WETH. Weth. https://etherscan.io/token/0xc02aaa39b223fe8d0a0e5c4f27ead9083c756cc2, 2023.

[64] Wikipedia. Sha-3. https://en.wikipedia.org/wiki/SHA-3, 2023.

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

[66] Valentin Wüstholz and Maria Christakis. Harvey: A greybox fuzzer for smart contracts. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1398–1409, 2020.

[67] Yinxing Xue, Mingliang Ma, Yun Lin, Yulei Sui, Jiaming Ye, and Tianyong Peng. Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 1029–1040, 2020.

[68] Yinxing Xue, Jiaming Ye, Wei Zhang, Jun Sun, Lei Ma, Haijun Wang, and Jianjun Zhao. xfuzz: Machine learning guided cross-contract fuzzing. *IEEE Transactions on Dependable and Secure Computing*, 2022.

[69] ycharts. Ethereum market cap. https://ycharts.com/indicators/ethereum_market_cap, 2022.