Confusum Contractum:
Confused Deputy Vulnerabilities in Ethereum Smart Contracts

Fabio Gritti†, Nicola Ruaro†, Robert McLaughlin†, Priyanka Bose†, Dipanjan Das†, Ilya Grishchenko†, Christopher Kruegel†, and Giovanni Vigna†

†University of California, Santa Barbara
{degrigis,ruaronica,robert349,priyanka,dipanjan,grishchenko,chris,vigna}@cs.ucsb.edu

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 inter-communication. 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 JACKAL, a system that is able to automatically identify and exploit confused contracts and candidate target contracts on the Ethereum mainnet. We leveraged JACKAL to analyze a total of 2,335,193 smart contracts deployed in the past two years, and we identified 529 potential confused contracts for which we were able to generate 31 working exploits. When investigating the impact of our exploits, 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 [20] – the most popular blockchain platform for dApps – from 51 billion to 568 billion dollars in just one year (Nov. 2020 - Nov. 2021) [82]. Despite the recent market correction, which saw Ethereum’s market cap drop to 201 billion dollars [66], 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) [77]. 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 (~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) [21], 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) [19] 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 [59]. 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 [60] 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 [12].

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 [55], and re-entrancy [7], via specific checks ensuring single entrance [54]. In addition, these and other “traditional” vulnerabilities have been the target of many automated verification and bug-finding solutions [4, 5, 13, 28, 31, 37, 39, 48, 57]. 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 [58], and led to a financial loss of $610 million in August 2021 [62]. 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 JACKAL, which is a first step in automatically detecting such flaws.

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 show the impact of our findings by producing 31 working exploits with the potential of jeopardizing 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 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 [20], are known to be “programmable blockchains.” Specifically, together with the standard currency-bearing user accounts [22], 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 [21]) 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 [1], or Vyper [70]). Once compiled, smart contracts are stored on-chain in the form of bytecode and executed on-demand by EVM.

2.3 Smart Contract Execution

While the execution state of a smart contract is defined by many elements [36], 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 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 [77]) 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 [18]).

The execution of a smart contract in the EVM is Turing-complete: the EVM can perform any computation as long as the user initiating the execution has sufficient ETH [21] to pay the required gas fee (note that there is an upper limit to the gas fee that can be specified [23]). This mechanism is a defense against denial of service and general abuse of resources [9].

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 [76] 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 [77] 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).

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

argSize: size of the CALLDATA located in memory at the argOffset.

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

retSize: size of the data returned by the callee (located in memory at retOffset).
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.

Scenario (1) 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. Scenario (2) 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.

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 flex-
contract TradingBot {
    struct Op { address target; bytes _calldata; }
    function execute(Op[] memory ops) public {
        uint i;
        for (i = 0; i < ops.length; i++)
            ops[i].target.call(ops[i]._calldata);
    }
}

Figure 2: Solidity code of the TradingBot. The execute function receives a list of tuples (Line 3) that are later used to make flexible, external function calls (in Line 6).

contract xyzToken {
    mapping (address => unit) public vault;
    function transfer(address dst, uint256 val) public {
        address src = msg.sender;
        require(vault[src] >= val);
        vault[src] = sub(vault[src], val);
        vault[dst] = add(vault[dst], val);
        return true;
    }
    function balanceOf(address addr) public {
        return vault[addr];
    }
}

Figure 3: Solidity code of a contract implementing a custom crypto-currency. The function transfer, provided by the standard ERC20 [52] 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.

To be able 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 3). 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 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:

(R1) A successful attack needs a pair of contracts: a confused contract \( C_c \) and a target contract \( C_t \). The 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 contract attack. In fact, a STATICCALL does not allow a $C_1$ 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 favor of DELEGATECALL [17]; in addition, it is used in less than 0.02% of the contracts in our dataset.

It is worth noting that a confused contract attack is fundamentally different from the traditional exploitation of access control vulnerabilities in smart contracts [5, 29, 43]. Given a contract, we are neither looking for functions with missing access control checks, nor are we trying to find a way to bypass insufficient access control checks present in the contract. Rather, in a confused contract attack, our goal is to satisfy the access control checks in a $C_1$ by “borrowing” the identity of the expected contract (i.e., the confused contract) through a controllable CALL within its code. In fact, when we run tools that look for access control vulnerabilities, such as AChecker [29], against real-world contracts equivalent to the TradingBot (Figure 2) and the xyzToken (Figure 3), no warnings are raised.

To demonstrate the fundamental difference between the exploitation of classic access control vulnerabilities and confused contract vulnerabilities, consider the contract in Figure 5. NFTGamble allows users to participate in a 

![Diagram of a confused contract attack](image-url)

**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.
and some bytes (callbackdata) used in a callback to notify gamblers in case they win (Line 17). By registering this callback, a gambler can implement further advanced logic (such as automatically re-selling the NFT) once the notification is received. If a user happens to be a winner (decided by the function check_win at Line 14), the contract transfers the ownership of the current NFT (NFT_ID) to the gambler address (Line 16). Immediately after that, the contract notifies the gambler by performing a call using the callbackdata specified during the function invocation. The contract has two distinct and unrelated vulnerabilities.

Access Control Vulnerability. The first vulnerability is a missing access control check in the changeOwner function (Line 6). In fact, anyone can call this function and change the _owner variable, thus gaining the privileges to withdraw all the fees collected by NFTGamble (Line 10). This vulnerability can be identified by tools such as AChecker [29], but it is not considered within the scope of JACKAL.

Confused Contract Vulnerability. An attacker can exploit the contract by calling the function gamble with the gambler parameter equal to `<NFTMARKET>`, and the callbackdata that will trigger a malicious call to the setApprovalForAll function [56] in the target NFT Market contract. Practically, when the NFTGamble contract will execute the callback at Line 17, it will perform a CALL to NFTMARKET.setApprovalForAll(<ATTACKER>, True), giving the attacker administrative privileges over all the NFTs owned (by NFTGamble) within the NFT Market contract. This is an instance of a confused contract vulnerability that is detected using our proposed approach and would be instead missed by systems like AChecker [29].

From this example, it is also clear that while most access control vulnerabilities can be identified by looking at a single contract, confused contract vulnerabilities exist because of the (trust) relationships between multiple contracts, and, as a result, these vulnerabilities require analyses that take into account multiple contracts and the way in which they interact.

4 Approach

To identify the requirements for a confused contract attack on the blockchain (R1-3 discussed in Section 3.2), we developed JACKAL. Figure 6 presents an overview of our system. In the following paragraphs, we describe each step of our analysis.

1 Bytecode Lifting and CFG Construction. To begin, JACKAL 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 [31].

2 Call Inspector. Once the CFG is built, JACKAL 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, JACKAL first statically verifies that both arguments are not constant (if they are constant, they cannot be controlled by the attacker). Then, JACKAL 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, JACKAL 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, JACKAL 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, JACKAL 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 (Cc).

3 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.

4 Checkers. 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 Cc-Cc pairs where the confused contract Cc has any special privilege to modify the target contract Ct’s storage. Similarly, the Token Checker finds target contracts where the confused contract holds (or held in the past) any balance of digital assets.

5 6 Attack Generation. Finally, JACKAL attempts to automatically synthesize an exploit for the warnings generated by the Token Checker, while the ones produced by the Generic Checker are passed to an analyst to verify whether it is possible to create an attack.

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 1). 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 1 (Figure 6) 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 [63] with an extension proposed by Falke et al. [25]. In the following, we introduce a few important features (F) of our symbolic execution engine.

(F1) Directed Exploration. We employ directed symbolic execution [49] toward previously-identified target CALL instructions, using the inter-procedural CFG provided by 1. 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 number1. 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 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 [77]. 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

---

1To facilitate reproducibility, we fix an arbitrary reference block number 16380000. We discuss this choice in Section 6.
the KECCAK256 hash of the $S$ bytes starting at the target offset. This cryptographic operation cannot be symbolically analyzed [14]. For this reason, different strategies were proposed to address this challenge [6, 28, 39, 47, 57]. In our symbolic execution engine, we use the approach proposed by Frank et al. [28]. Whenever we obtain solutions for the CALL 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 [16], 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 does not follow calls to external functions (in other contracts), and we use partially concrete storage and execution context. Hence, in some cases, JACKAL 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 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 [24] to execute the confused contract’s bytecode, supplying the concrete CALLDATA value generated by the Call Inspector (Figure 6, 2) 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 may already be 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 Cc and a target contract Ct 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 [52, 53]. 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 confused contract Cc 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 Cc 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 Cc, 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.

Ultimately, both our checkers produce as output a tuple (Cc, Ct, B), reporting that there exists a confused contract Cc associated with a possible target Ct that might be exploitable at block B.

4.4 Attack Generation

At this step, JACKAL attempts to craft an exploit for the warnings raised by our checkers (Section 4.3).

Token Checker Exploits. We automate the exploit synthesis for the warnings generated by the Token Checker. Given a warning (Cc, Ct, B), to exploit Cc and transfer all the tokens owned in Cc to an attacker-controlled account (A), the attacker needs to find a CALLDATA input for Cc such that (1) the “open” CALL within the code of Cc executes the standardized transfer function [52] of Ct (the Token Contract [52, 53]) with the right arguments (i.e., from, to, value), and (2), the execution of Cc successfully terminates after returning from the call to Ct.

To generate the required CALLDATA, JACKAL leverages symbolic execution as discussed in Section 4.1.1, using block B as a reference block for our partial concrete storage (F2). In particular, JACKAL first uses directed symbolic execution (F1) to reach the CALL, adding extra constraints to the execution to properly set the arguments of the CALL: tAddr=Ct, tFunc=0xa9059cbb (the function id of transfer). Finally, the rest of the data sent to Ct should include the encoded arguments for transfer (the tFunc) such that: from=Cc, to=A, and value=0x1.

Then, JACKAL leverages the CFG of Cc to identify a reachable STOP/RETURN [77] instruction and resume directed symbolic execution in order to reach it.

Finally, JACKAL solves the path constraints and obtains the desired value for CALLDATA, which allows for the automatic synthesis of the exploit.

Generic Checker Exploits. Crafting an exploit for the warnings generated by the Generic Checker is currently a semi-automatic effort. In particular, we apply the same methodology as described for the Token Checker warnings, but in this case, we need to manually understand which arguments we need to provide to the target function (tFunc) to avoid an early failure (revert) during the execution of Ct. We further discuss this issue in Section 6.

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 [65] to parallelize our tasks.

Dataset Information. We extracted all smart contract addresses created in the 24-month period from December 2020 to December 2022. 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 7, 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

2 Corresponding to blocks in the range 11363270–16086235.
Table 1: Source code availability for smart contracts in our dataset considering unique and non-unique code.

<table>
<thead>
<tr>
<th></th>
<th>Has Source Unique</th>
<th>Has Source Total</th>
<th>No Source Unique</th>
<th>No Source Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small Size (&lt;5Kb)</td>
<td>58,869</td>
<td>1,113,287</td>
<td>54,933</td>
<td>912,392</td>
</tr>
<tr>
<td>Large Size (≥5Kb)</td>
<td>138,765</td>
<td>159,070</td>
<td>55,390</td>
<td>63,919</td>
</tr>
</tbody>
</table>

5.1 Confused Contracts

Static Analysis. We obtained CFG reconstruction and static analysis results by running Gigahorse [31] 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 8, 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.

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 must 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). As illustrated in Figure 9, most of the CALLs were analyzed in less than a minute, while our engine encountered a timeout for 29,400 CALL instances (3.77%). JACKAL 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. Every such CALL belongs to a unique contract that can potentially be vulnerable to a confused contract attack. To identify 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 with “open” CALL (confused contracts). This step took on average 1.71 seconds per contract and reduced the number to 529 confused contracts for which we can replay inputs that reach the CALL instruction. 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 returning from the call. Hence, false positives are still possible, e.g., the msg.sender might be checked after the call.

3A contract can contain multiple CALL opcodes.
we attempt to generate end-to-end exploits for the warnings we detected some potentially-sensitive persistent storage were observed when changing the \texttt{msg.sender}.

Token Checker. The \textit{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.

Note that there is no overlapping between the warnings generated by the two checkers. In fact, we excluded all the known token contracts (i.e., ERC20/ERC721 contracts) from the scope of the Generic Checker before running it.

5.3 Confused Contract Exploitation

In Section 5.2, we reported that \textsc{JackAL} found 84 confused contracts and associated target contracts. However, even if our tool successfully reported conditions for confused contract attacks, one important question remains: \textit{is it possible in practice to generate an exploit?} To answer this question, we attempt to generate end-to-end exploits for the warnings generated by our checkers. 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 some-times 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 [24], using as a reference block the one reported by our checkers in their warnings (i.e., a block in which a confused contract has a valid target contract).

Generic Checker Warnings. After looking into the 32 warnings for confused contracts generated by the \textit{Generic Checker}, we found that one of them interacted in the past with the Seaport [45] smart contract: an ERC-721 token marketplace. After a brief manual review (~5 minutes) of the Seaport contract and its functions, we were able to craft an exploit leveraging the same automatic approach used for the Token Checker Exploits (see Section 4.4), 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 10, 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 \texttt{CALL}’s arguments.

```cpp
// Cancel all orders from a given * offerer with a given zone in bulk * by incrementing a counter. *
// Note that only the offerer may increment the counter. */
function incrementCounter () external returns (uint256 newCounter);
```

Figure 10: The \texttt{incrementCounter} function in Seaport.

The developers explicitly noted that \textit{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 \texttt{msg.sender} after the controllable \texttt{CALL} instruction but before terminating its execution (and, thus, the contract reverts). In 2 other cases, we could not fully control the target \texttt{CALL} instruction because of the additional checks on the \texttt{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 \texttt{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 \texttt{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. The manual triaging of these 32
warnings took on average 10 minutes per instance.

**Token Checker Warnings.** The Token Checker identified 52 confused contracts. Our automatic exploit generation analysis was able to synthesize 30 working exploits. For each warning, this process took on average less than 45 minutes. The obtained exploits demonstrate the possibility of transferring the assets owned by a confused contract (for a specific token) to an account of an attacker’s choice. That is, all our exploits work by manipulating the input of the confused contract’s public function so that, in the arguments of the “open” CALL, the target address `tAddr` is the address of a token contract and the target function `tFunc` is the transfer function of the contract [52], which sends the assets of the confused contract to the attacker-specified address.

In Table 2, we show the results of our findings considering confused contracts that held a value greater than $100. 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 22 warnings for which we could not generate 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 19 of them, we were unable to set the `tFunc` to the right contract target.

<table>
<thead>
<tr>
<th>#</th>
<th>Token</th>
<th>Block</th>
<th>Span</th>
<th>Val</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DAI</td>
<td>14104828</td>
<td>146</td>
<td>$838,436</td>
</tr>
<tr>
<td>2</td>
<td>DAI</td>
<td>11469710</td>
<td>476</td>
<td>$190,090</td>
</tr>
<tr>
<td>3</td>
<td>BAS</td>
<td>11454250</td>
<td>2</td>
<td>$12,610</td>
</tr>
<tr>
<td>4</td>
<td>Hegic</td>
<td>11492550</td>
<td>42</td>
<td>$10,007</td>
</tr>
<tr>
<td>5</td>
<td>wETH</td>
<td>13574103</td>
<td>12,199</td>
<td>$2,404</td>
</tr>
<tr>
<td>6</td>
<td>wETH</td>
<td>15625577</td>
<td>14,867</td>
<td>$1,438</td>
</tr>
<tr>
<td>7</td>
<td>wETH</td>
<td>15645205</td>
<td>651</td>
<td>$1,405</td>
</tr>
<tr>
<td>8</td>
<td>wBTC</td>
<td>15200611</td>
<td>1</td>
<td>$993</td>
</tr>
<tr>
<td>9</td>
<td>LooksRare</td>
<td>14634155</td>
<td>1,927,641</td>
<td>$883</td>
</tr>
<tr>
<td>10</td>
<td>LUSD</td>
<td>15451410</td>
<td>62</td>
<td>$391</td>
</tr>
<tr>
<td>11</td>
<td>wETH</td>
<td>15596427</td>
<td>1146</td>
<td>$261</td>
</tr>
<tr>
<td>12</td>
<td>wETH</td>
<td>14170312</td>
<td>19,396</td>
<td>$133</td>
</tr>
</tbody>
</table>

**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`. `Span` represents the length of time (in blocks) during which the confused contract was exploitable (a new block is mined approximately every 12 seconds). `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 historical price per token using the Uniswap [69] contract.

**Ethical Considerations.** 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. 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. Finally, we notified the Cybersecurity and Infrastructure Security Agency (CISA) [11] regarding the high-impact vulnerabilities identified by JACKAL (Table 2).

### 5.4 Comparison with Existing Systems

We compare JACKAL with two state-of-the-art bytecode-only symbolic executors for smart contracts: Mythril (Github commit: d531d8b) [13] and ETHBMC (Github commit: e887f33) [28]. For this evaluation, we took all the 379,175 contracts that include a potentially vulnerable `CALL`.

**Mythril.** Mythril includes a security analysis that detects publicly reachable `CALL`s. However, their definition of controllable `CALL`s is broader than our `R2`. In fact, Mythril simply raises an alert when the `tAddr` of a `CALL` can be controlled by an attacker, saying nothing about `tFunc`. In total, Mythril flagged 33,182 `CALL`s (in 26,161 different contracts) as controllable. We investigated a random sample of 100 warnings and discover that 96 of them are false positives (given our threat model). That is, they are `CALL`s whose `tFunc` is not controllable by an attacker. Only 4 warnings are related to fully controllable `CALL`s (i.e., both `tAddr` and `tFunc` are controllable). Of these, 3 of them have also been identified by JACKAL, while one is a false negative of our system. In particular, this false negative is a consequence of selecting one single entry point for our symbolic execution as discussed in Section 6.

Finally, Mythril identified controllable `CALL`s in only 445 of the 529 (84.12%) contracts identified by JACKAL. These are false negatives of Mythril’s analysis.

**ETHBMC.** ETHBMC includes a security analysis that raises an alert when it finds a `CALL` with attacker-controllable `tAddr` and `value` (see StealMoney threat model [28]). Note that this captures only a subset of controllable `CALL`s that satisfy our `R2`. Thus, for a more complete and fair comparison, we evaluated ETHBMC’s analysis under two strategies: (1) we log all the `CALL`s with controllable `tAddr` as soon as the tool observes them, and (2) by analyzing the final reports. When considering (1), we gather a total of 14,159 warnings for controllable `CALL`s (in 7,603 different contracts). We sampled 100 of these warnings and found 13 `CALL`s that are fully controllable (`R2`). JACKAL missed 5 of these `CALL`s (and
found the other 8). Again, we found that the root cause for our false negatives is related to the limitation of selecting a single entry point (see Section 6) for our symbolic execution. The remaining 87 CALLs are false positives: 43 are STATICCALL with controllable tAddr (that are not part of our threat model), 34 are CALLs with tFunc arguments that cannot be controlled, and 10 are unreachable CALLs, i.e., there are checks on the msg.sender address before reaching the CALL.

When just considering the final reports, ETHBMC reported a total of 196 warnings for controllable CALLs (in 128 different contracts). Notably, this new set of warnings contains only 8 of the 529 confused contracts flagged by JACKAL.

Overall, considering both evaluation strategies, ETHBMC identified only 337 (63.7%) of the 529 confused contracts flagged by JACKAL after step 3.

Our evaluation shows that while both tools are able to identify public reachable, and, to some extent, controllable CALLs, none of the warnings generated by the tools provide additional insights regarding existing target contract con-trols with a contract with a controllable CALL (R3).

In closing, we would like to emphasize that the purpose of this evaluation is not to demonstrate the superiority of our symbolic execution engine. Instead, we want to show that existing tools do not directly detect confused contract vulnerabilities. One would have to implement additional analysis steps and filters to address the high number of false positives and false negatives. Of course, these modifications could have been implemented on top of either Mythril or ETHBMC. However, for this work, we decided to build our analysis on top of Gigahorse, given its high-quality results in CFG reconstruction and contract decompilation [31].

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 [31]. While it is true that imprecision in the CFG analysis would lead to both false positives/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. JACKAL could be extended 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 1638000 throughout our analyses for the identification of R1 and R2. While running our analyses on each block would be ideal, it may be impractical. Moreover, the benefits of fixing an arbitrary reference block are twofold. First, this allows for the reproducibility of our results. Second, choosing a recent block allows us to check whether 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 CALLEDATA supplied by the attacker and the values of the CALL arguments, which corresponds to our attacker model described in Section 3.2.

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

Attack Generation. Our system is able, under certain circumstances, to automatically generate an exploit for the warnings raised by the Token Checker. However, the complexity of a contract’s logic can sometimes hinder this process (e.g., the path constraints collected during symbolic execution are too hard to solve). In these cases, finding the right CALLEDATA input to trigger a confused contract attack can be highly time-consuming, thus, we cannot confirm the exploitability.

Currently, the warnings generated by the Generic Checker require manual work to identify the right arguments to supply to the tFunc in a target contract. While this process could be automated, we decided to leave it as a future work considering the low complexity of the warnings raised by this checker during our Evaluation (Section 5).

7 Related Work

7.1 Confused Deputy Vulnerabilities

Since its introduction by Norm Hardy [33], the concept of confused deputy vulnerability has been driving many research efforts. For instance, this concept has been leveraged in research efforts looking at privilege escalation attacks in Android and AppleOS [73, 78]. To the best of our knowledge, our work is the first to apply the confused deputy bug class to model attacks on smart contracts.

7.2 Smart Contract Analysis

Static Analysis. Grech et al. proposed Gigahorse [31], 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 [5, 30, 40, 64]. A similar approach has been proposed by Fesit et al. with Slither [26], a static analysis framework that targets Solidity [1] source code and supports the detection of a multitude of different bug classes.
Tikhomirov et al. proposed SmartCheck [67], a system that translates Solidity source code into an XML representation and identifies bugs with the help of XPath patterns. Tsankov et al. proposed Securify [68], a framework that leverages a smart contract’s dependency graph to identify patterns of vulnerabilities. Finally, SmartDagger [41] and Clairvoyance [80] are static analysis systems that focus on discovering traditional vulnerabilities (e.g., reentrancy, timestamp manipulation, and DoS) afflicting inter-contract interactions.

Symbolic Execution & Bounded Model Checking. The first effort in this area has been proposed by Luu et al. with Oyente [47], followed by Krupp et al. [39] 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 Mythril [13] and Manticore [50] have been proposed and became the de-facto standard tools in the arsenal of professional smart contracts auditors. Ma et al. proposed Pluto [48], 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 [28], a bounded model checker that supports the symbolic execution of a smart contract bytecode even when external interactions are present. Finally, Bose et al. proposed Sailfish [4], a symbolic execution system for automatically finding state-inconsistency bugs.

Fuzzing. Many systems based on fuzzing techniques have been developed through the years. Some of them require Solidity source code: Echidna [32], ContraMaster [71, 72], Foundry [27], fuzzing-like-a-degen [2], ChainFuzz [8], EXGEN [38], and xFuzz [81]. Others can work solely with a contract’s bytecode: Harvey [79], ContractFuzzer [37], ILF [34], Reguard [42], sFuzz [51], Smartian [10] and EF5-CF [61].

While JACKAL leverages ideas from prior static analysis and symbolic execution systems (that inspect individual contracts), our goal is different. First, we aim to precisely describe confused contract vulnerabilities, a new blockchain bug class that revisits the classic confused deputy vulnerability [33], but in the context of interactions between smart contracts. Second, we propose an approach that relies on different program analysis techniques to detect such flaws and show that they can be practically exploited (leading to considerable impact on the overall blockchain ecosystem).

8 Conclusions

In this paper, we introduced the confused contract vulnerability class, a variant of the traditional confused deputy bug applied 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 asset transfers from the confused contract to an attacker.

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

To identify opportunities for confused contract attacks in the blockchain, we developed two checkers that automatically identify target contract candidates. Based on these findings, we were able to develop 31 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.

9 Acknowledgements

We would like to thank our anonymous shepherd and reviewers for their valuable feedback. This material is based upon work supported by Google Security, Privacy, and Anti-Abuse Award. Any opinions, findings, conclusions, or recommendations expressed in this publication, are those of the author(s) and do not necessarily reflect the views of Google.

References


