AChecker: Statically Detecting Smart Contract Access Control Vulnerabilities

Asem Ghaleb
University of British Columbia
Vancouver, Canada
aghaleb@alumni.ubc.ca

Julia Rubin
University of British Columbia
Vancouver, Canada
mjulia@ece.ubc.ca

Karthik Pattabiraman
University of British Columbia
Vancouver, Canada
karthikp@ece.ubc.ca

***Abstract*—As most smart contracts have a financial nature and handle valuable assets, smart contract developers use access control to protect assets managed by smart contracts from being misused by malicious or unauthorized people. Unfortunately, programming languages used for writing smart contracts, such as Solidity, were not designed with a permission-based security model in mind. Therefore, smart contract developers implement access control checks based on their judgment and in an ad-hoc manner, which results in several vulnerabilities in smart contracts, called access control vulnerabilities. Further, the inconsistency in implementing access control makes it difficult to reason about whether a contract meets access control needs and is free of access control vulnerabilities. In this work, we propose *AChecker* – an approach for detecting access control vulnerabilities. Unlike prior work, *AChecker* does not rely on predefined patterns or contract transactions history. Instead, it infers access control implemented in smart contracts via static data-flow analysis. Moreover, the approach performs further symbolic-based analysis to distinguish cases when unauthorized people can obtain control of the contract as intended functionality.**

We evaluated *AChecker* on three public datasets of real-world smart contracts, including one which consists of contracts with assigned access control CVEs, and compared its effectiveness with eight analysis tools. The evaluation results showed that *AChecker* outperforms these tools in terms of both precision and recall. In addition, *AChecker* flagged vulnerabilities in 21 frequently-used contracts on Ethereum blockchain with 90% precision.

***Index Terms*—Smart contract, security, access control, data-flow analysis**

I. Introduction

Smart contracts are small programs deployed and running on top of a blockchain. Several blockchain networks, such as Ethereum [1] and Cardano [2], support running smart contracts, with Ethereum currently being the most popular blockchain platform. Ethereum smart contracts are written in high-level Turing-complete languages, e.g., Solidity [3], and get compiled to low-level EVM bytecode that is deployed to the blockchain. The bytecode deployed on Ethereum permissionless public networks is visible to the public, and can be invoked by anyone who has an account on the blockchain [4].

As smart contracts can be called by any user on the blockchain, smart contract developers typically implement access control to manage who can call specific functions within the contract or execute critical actions, such as withdraw money or destroy the contract and remove it from the blockchain. Unfortunately, Solidity was not designed with a permission-based security model in mind. Therefore, smart contract developers implement access control checks in an ad-hoc manner, such as through the use of language-special-constructs, e.g., function modifier, or assertions and conditional statements, e.g., *require(access-control-condition)*.

Failing to implement access control checks properly can result in either (1) weak checks that can be bypassed by unauthorized users, or (2) missing access control for code statements that need to be protected. We call these *access control vulnerabilities*. There have been many instances of real-world attacks due to access control vulnerabilities. For example, in May 2021, an attack targeting a smart contract used by the ValueDeFi platform [5] led to the loss of about 10M. *The attack was due to a mistake in a single line of code that resulted in an access control vulnerability, which enabled the attacker to make themselves an owner of the contract.* Another example is the hack that targeted Parity Wallet and led to locking out (freezing) about 280M worth of Ether [6]. The attack was due to having an unprotected function in the contract, which enabled the attacker to reset the variables storing addresses of wallet owners and destroy the contract, thereby freezing all the funds in the wallet.

Static analysis can help identify access control vulnerabilities before the deployment of the contract on the blockchain. This is important as it is difficult to modify a smart contract after its deployment, and transactions are immutable; hence losses cannot be rolled back. Unfortunately, statically analyzing smart contracts for access control vulnerabilities is a challenging problem because: (1) The lack of access control policy specifications make it difficult to precisely identify access control checks. (2) Smart contracts may contain code patterns that look like vulnerabilities but are really part of the contracts' functionality, and it is not straightforward to statically distinguish these patterns from vulnerabilities due to the lack of the contract access control specifications.

The most relevant prior work on detecting access control vulnerabilities are Ethainter [7] and SPCon [8]. Ethainter performs information-flow analysis to detect five types of access control vulnerabilities. It relies on pre-defined rules to identify the access control checks, assuming contracts are written in a specific style, and compilers always generate certain bytecode patterns for these access control checks. Our results show that such pre-defined rules do not generalize well, making Ethainter fail on a wide range of samples. Further, the

pre-defined patterns for access control checks are not precise enough and thus result in false alarms.

Unlike Ethainter, SPCon does not rely on specific code patterns but rather analyzes smart contracts for permission bugs through mining access control roles using the contract transaction history available on the blockchain. To mine access control roles, SPCon assumes that the analyzed contract have transaction record for the various contract functions; however, not all contract functions get called and have transactions. This is because these functions only get called in specific circumstances, e.g., to remove smart contracts from the blockchain. Further, SPCon assumes each transaction is benign and is performed by an authorized account of the function, as per the desired contract security policy. However, this cannot be guaranteed, especially for smart contracts having access control vulnerabilities.

Besides SPCon and Ethainter, other approaches [9]–[14] target various types of smart contract bugs, including vulnerabilities due to missing access control. However, they do not focus on contract-wide access control vulnerabilities resulting from weak and missing access control but rather analysis missing access control for specific code statements, mostly based on pre-defined access control patterns.

In this work, we propose an efficient approach, *AChecker* (Access Control Checker), for discovering access control vulnerabilities in smart contracts. Our key insight for identifying access control checks is that they have unique functionality that can be inferred by analyzing data dependencies [15] in the conditions forming the checks. After identifying access control checks, we detect access control vulnerabilities by analyzing data dependencies between the data inputs from the contract users and (a) state variables storing access control data and (b) a set of predefined code statements. The main intuition behind our analysis is that these "critical" instructions can only be manipulated by trusted users, and thus should be protected by access control checks. That is, we formulate the detection of access control vulnerabilities as a taint analysis problem [16], without relying on pre-defined access control patterns or existing transactions.

Furthermore, to avoid reporting false-positive results, our analysis identifies cases of *potentially intended behaviors*. We assume a behavior is intended when the code implements non-access control constraints that allow manipulating the access control data only under specific conditions. Our intuition behind this heuristic is that having non-access control constraints to guard access control data *under specific conditions* implies that the developer intentionally implemented this code behavior. We employ symbolic execution analysis to synthesize constraints under which manipulation of access control data occurs; we then separate potentially intended behaviors from more certain access control vulnerabilities.

To the best of our knowledge, AChecker *is the first technique that statically reasons about access control checks without requiring either pre-defined code patterns or pre-existing transactions history. Further, the intended behavior is reported by the current analysis tools as vulnerabilities, and* AChecker *is the first to optimize for high-precision by analyzing for intended behavior cases and separating them as potentially intended cases, thereby minimizing false alarms.*

We evaluate *AChecker* on three datasets collected by prior work: the first one, CVE [8], consists of 15 contracts with access control vulnerabilities; the other two datasets, SmartBugs [17] and Popular-contracts [18], are large dataset of real-world Ethereum contracts, with 47,518 and 3,000 contracts, respectively. For our evaluation, we assess the efficiency of *AChecker* by comparing it to that of eight existing tools that target access control vulnerabilities, namely SPCon [8], Ethainter [7], Securify [9], Manticore [10], Maian [11], Mythril [12], Slither [13], and Smartcheck [14].

We summarize our main contributions below.

- Propose a novel static data-flow-analysis-based technique for efficient identification of access control checks, relying on neither pre-defined code patterns nor existing transactions history, unlike other tools.
- Propose a novel symbolic execution-based approach for reducing false-positives by automatically inferring cases where untrusted users are allowed to manipulate access control data as an intended behavior of the contract.
- Combine our proposed methods in a consolidated approach, *AChecker*, for detecting access control vulnerabilities. We implement the approach in an automated tool and make the implementation publicly available [19].
- Evaluate *AChecker* on three public datasets and show that it outperforms all eight existing approaches used as a baseline, in terms of both precision and recall. Furthermore, it is able to detect a large number of vulnerable contracts in SmartBugs and Popular-contracts datasets with a high precision.

II. Motivating Examples

In this section, we use real-world examples to discuss different types of access control vulnerabilities and the challenges of finding them in smart contracts. Moreover, we discuss cases of potentially intended behavior in the contract that are reported as access control vulnerabilities by current analysis tools.

Our definition of access control vulnerabilities is based on prior work [7]–[14]. We classify access control vulnerabilities into two main categories, according to their cause. The first group involves vulnerabilities that occur due to vulnerable access control checks that can be bypassed by attackers. We refer to these as "*violated access control checks (VACC)*". The second group consists of the vulnerabilities that arise due to the lack of access control for critical instructions. We refer to these as "*missing access control checks (MACC).*"

A. Violated Access Control Check (VACC)

The code in Fig. 1 is simplified from a real-world contract that implements an Ethereum token for Business Alliance Financial Circle (BAFC) [20]. The contract was disclosed as having an access control vulnerability and is assigned the CVE "CVE-2018-19830". In this contract, the `owner` state variable (defined in line 2) is used to store the address of the owner

of this contract, and the `frozenAccount` (line 3) is a state mapping that stores frozen accounts that cannot perform specific actions. The modifier[1] `onlyOwner` (lines 5-11) checks if the caller is the owner, and the modifier `unFrozenAccount` (lines 12-15) checks if the address of the caller is not frozen. The `onlyOwner` modifier is used to protect the functions `freezeAccount`, `switchLiquidity`, and some others (not shown in Fig 1). The `unFrozenAccount` modifier is used in several functions, such as `transfer`.

```
1 contract BAFCToken {
2   address owner = msg.sender;
3   mapping (address => bool) public frozenAccount;
4   /****** modifiers ******/
5    modifier onlyOwner {
6       if (owner == msg.sender) {
7           _; }
8       else {
9           InvalidCaller(msg.sender);
10          throw;}
11   }
12   modifier unFrozenAccount{
13      require(!frozenAccount[msg.sender]);
14      _;
15  }
16 /****** Functions *******/
17  function UBSecToken () public {
18     owner = msg.sender;
19     totalSupply = 1.9 * 10 ** 26;
20  }
21  function freezeAccount(address target, bool freeze)
        onlyOwner public {
22      frozenAccount[target]=freeze;
23    }
24  function switchLiquidity(bool _transferable) onlyOwner{
25      transferable=_transferable;
26  }
27  function transfer(address _to, uint _value)
        unFrozenAccount onlyTransferable {
28    if (frozenAccount[_to]) {
29       InvalidAccount(_to, "Frozen receiver account");}
30    else {
31      balances[msg.sender]=balances[msg.sender].sub(
            _value);
32      balances[_to] = balances[_to].add(_value);
33    }
34  }
```

Fig. 1. Real-world contract with violated access control checks.

The function `UBSecToken` (line 17) is used to initialize the `owner` variable and some other data items. This function is supposed to be defined as a constructor to get executed when the contract is deployed to initialize the `owner` with the address that creates (deploys) the contract. As a constructor, it would not be part of the code running on the blockchain. However, the developer made a mistake and named the constructor `UBSecToken` instead of `BAFCToken`; hence, the misspelled constructor leads to the `UBSecToken` function to be callable by anyone. This mistake enables anyone to make themselves an owner of this contract, and perform actions such as switching the status of the token liquidity using the `switchLiquidity` function (line 24). Furthermore, the user who hijacks the ownership of this contract can freeze/unfreeze any address having tokens managed by the contract, thereby violating the access control check `unFrozenAccount` and becoming able to control who can perform some actions, such as transferring or receiving tokens using function `transfer` (line 27).

We found that Ethainter [7] does not report vulnerabilities in this contract because Ethainter's inference rules do not recognize the access control check implemented by the `onlyOwner` modifier. The code of `onlyOwner` was compiled to a bytecode pattern that does not match the style expected by Ethainter's inference rules due to an extra jump block in the bytecode. Although this case can be fixed by adding more rules, it is difficult to enumerate all such scenarios generated by the compiler, and add inference rules for them.

Similarly, this contract is out of the scope of SPCon [8] as there are only limited transactions on this contract's functions, and as discussed earlier, SPCon mines access control rules from transactions. Thus, SPCon was not able to mine the security policy of this contract to be able to detect the vulnerability. In addition, for scalability, SPCon only considers two transactions executed in sequence to trigger vulnerabilities; thus, it cannot detect access control checks attacked through more than two transactions, such as `unFrozenAccount` getting violated when `onlyOwner` gets violated.

Finally, to the best of our knowledge, no approach finds the mapping between the violated access control checks and the affected functions in the contract; hence, existing tools do not find the contract function(s) that becomes accessible to attackers when an access control check is violated and leave that for developers to check it manually, which is difficult and time-consuming for complex contracts. For example, in the contract in this example, none of the existing tools report that the functions `freezeAccount`, `switchLiquidity`, `transfer`, etc., are accessible by attackers because `onlyOwner` is violated in function `UBSecToken`.

B. Missing Access Control Check (MACC)

As mentioned earlier, the contract code may contain critical instructions that have to be protected, so they do not get executed or manipulated by attackers [21]–[23]. For example, in the code excerpt shown in Figure 2, taken from a real-world contract [24], the instruction `selfdestruct(msg.sender)` at line 2 will remove the contract from the blockchain and send whatever amount is in the contract balance to the address provided as a parameter to the instruction (`msg.sender`). Therefore, only authorized users, e.g., owner, should be allowed to invoke the function `removeContract`. Further, the parameter of the instruction `selfdestruct` should not be set by unauthorized users.

```
1 function removeContract() public {
2  selfdestruct(msg.sender);//remove contract from network
3 }
```

Fig. 2. Missing access control check example.

Several existing tools cannot detect the two vulnerabilities in this example. For instance, the function `removeContract` is called only once to destroy the contract, and hence SPCon cannot detect these vulnerabilities because transactions to the function `removeContract` (needed for SPCon to mine

[1]Function modifiers check conditions prior to the functions' execution.

```
1  modifier onlyOwner() {
2   require(owner()==_msgSender(),"Caller is not owner");_;
3   }
4  function owner() public view returns (address) {
5    return _owner;
6   }
7  //@dev Destroy contract and reclaim leftover funds
8  function kill() external onlyOwner {
9    selfdestruct(payable(_msgSender()));
10  }
```

Fig. 3. Protected `selfdestruct` reported as a vulnerability by prior work.

```
1  address public owner;
2  uint256 constant howMuchToBecomeOwner= 1000 ether;
3  function changeNameSymbol(string _n, string _s)external{
4     if (msg.sender==owner){
5       /* omitted code */
6   }
7  function changeOwner(address _newowner) payable external{
8     if (msg.value>=howMuchToBecomeOwner){
9        owner.transfer(msg.value);
10       owner=_newowner;}
11  }
```

Fig. 4. Intended behavior example.

security roles) will only be available on the blockchain when the attack is done and the contract is destroyed. On the other hand, approaches relying on pre-defined patterns report high false-positives when analyzing for these vulnerabilities. For example, in the code in Figure 3, taken from an NFT contract called "Sugoi NFT NYC" [25], Ethainter flags the function `kill`, having `selfdestruct` (line 9), as vulnerable even though it is protected by a secure access control modifier `onlyOwner`. We found that Ethainter fails to recognize the modifier `onlyOwner` protecting the function `kill`. This is because `onlyOwner` calls the `owner()` function to obtain the address of the owner, and Ethainter's inference rules cannot match patterns across functions. Ethainter looks for patterns of access checks within connected code blocks; however, calling another function from within the access check results in jumping into other blocks to execute the function, which makes it difficult for Ethainter to identify the access check.

C. Potentially Intended Behaviors

In some cases, the contract code may be intentionally implemented to allow any user to update access control state variables under specific conditions. Analysis tools should distinguish these cases from exploitable true vulnerabilities to maximize benefit of the analysis tools. We refer to these cases as *potentially intended behaviors*. For example, in the code excerpt in Figure 4, extracted from a real-world contract, the function `changeNameSymbol` has an access control check at line 4, so that only the owner of the contract, whose address is stored in the `owner` state variable, can call it. On the other hand, the contract has the function `changeOwner`, which enables anyone to buy the contract's ownership for 1000 Wei – Wei is the smallest denomination of the Ether cryptocurrency.

We find that almost all the existing tools looking for untrusted writes of access control data will report the `changeOwner` function as a vulnerability since anyone can change the `owner` state variable. However, this behavior is intentional: the contract owner intentionally allowed anyone to buy the ownership. When the paid amount gets transferred to the current owner, the `owner` is set to the buyer's address.

III. *AChecker* Approach

A. *Overview of* AChecker

Figure 5 shows the workflow of the proposed approach, *AChecker*. The approach consists of three major steps. (1) *AChecker* takes as input the contract bytecode, and performs static data-flow analysis to identify the access control checks that are implemented and used in the contract, as well as the corresponding state variables storing access control data. *AChecker* also identifies critical instructions that should be protected. (2) Using the identified access control state variables and critical instructions (from the previous step) as sinks, *AChecker* explores possible taint paths from user inputs (taint sources) to the sinks. (3) *AChecker* symbolically executes paths of the taint-flows reaching sinks to filter out infeasible paths and separate intended behavior cases from exploitable true vulnerabilities. Finally, *AChecker* flags the found vulnerabilities, and the corresponding functions to the users. We explain the steps in the following sub-sections.

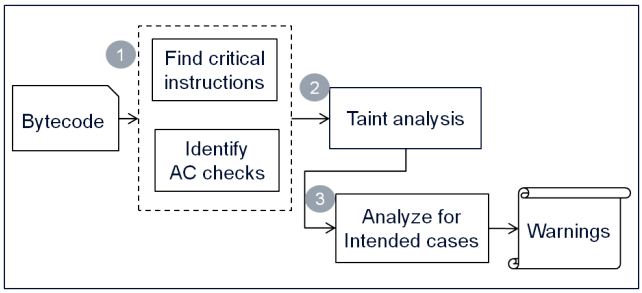

Fig. 5. *AChecker* workflow.

B. Identifying Access Control Checks

To find access control vulnerabilities, *AChecker* first identifies access control checks used in the contract, without relying on syntactic patterns of access control checks. Before discussing our approach for identifying access control checks, we formally define the following based on the functionality provided by access control checks in smart contracts:

Definition 1 (Access Control Check)**.** *Let C be a condition of a conditional control statement in a smart contract S, and C validates the callers of the contract S that can execute the code protected by the condition C, then the condition C is an access control check.*

Definition 2 (Access Control State Variable)**.** *In an access control check C, if C compares the caller of the contract against value(s) stored in a storage data item D or if C uses the contract caller to load a value from a storage data item D to evaluate C, then D is an access control state variable.*

Definition 3 (Authorized User)**.** *In an access control check C that uses an access control state variable D, any user whose address* $\in$ *D is an authorized user to execute the contract code protected by the access control check C.*

For example, in the contract code in Figure 1, the function `switchLiquidity` has the modifier `onlyOwner`. This modifier has the statement `if(owner==msg.sender)` that checks if the address of the caller (`msg.sender`) is equal to the address stored in the state variable `owner`. The execution of the function `switchLiquidity` happens only when the `if` condition is true. Thus, in function `switchLiquidity`, the condition `owner==msg.sender` is an access control check, the `owner` is an access control state variable, and the user whose address stored in the state variable `owner` is an authorized user to execute the function `switchLiquidity`.

The definitions mentioned above state that, the unique functionality of all access control checks, regardless of how they are written, is to check the callers of the smart contract against the access control state variables (representing various authorized users). Thus, our intuitive idea to identify access control checks in the contract code is to analyze for conditions that reference the contract caller and either compare it to values loaded from the contract storage (i.e., access control state variables) or use it to load values from the contract storage to evaluate the condition. In smart contracts, a built-in global variable is used to obtain the contract caller – `msg.sender` in the source code and `CALLER` in the bytecode. Therefore, we can leverage this feature to perform our analysis.

Algorithm 1: Identifying Access Control Checks

```
   Input: C ▷ Conditions
   Output: AC, SV ▷ Access control checks and state variables
1  begin
2      AC ← ∅ ▷ Initial set of access control checks
3      foreach condition c ∈ C do
4          flow ← BackwardAnalysis(c)
5          if CALLER & SLOAD ∈ flow then
6              s ← SlotOf(SLOAD)
7              if s ∉ mappings then
8                  AC ← AC ∪ c; SV ← SV ∪ s
9              else if s ∈ mappings & TypeValue(s) ∈
                Boolean then
10                 AC ← AC ∪ c; SV ← SV ∪ s
11             else if s ∈ mappings then
12                 t ← ForwardAnalysis(c,s)
13                 if not t then
14                     AC ← AC ∪ c; SV ← SV ∪ s
15     return AC, SV
```

We conduct our analysis to identify access control checks by statically analyzing data-flows of the conditions in the contract to obtain conditions that have data dependency on the two instructions, `CALLER` and `SLOAD` (storage load), as shown in Algorithm 1. The algorithm takes as input the conditions in the contract code and returns as output, the access control checks implemented in the contract and the corresponding access control state variables. For each condition in the contract, *AChecker* first performs backward data-flow analysis starting at the condition (lines 3-4). By considering all data dependencies, the resulting flows contain all instructions that are involved in evaluating the condition. Then *AChecker* filters conditions having data dependency on `CALLER` and `SLOAD` instructions (line 5). These filtered conditions form an overapproximation of access control checks.

For example, in the code in Figure 1, the modifier `onlyOwner` (lines 5-11) is used as access control to protect the function `switchLiquidity`. The simplified bytecode of this function, in static single assignment (SSA) form, is shown in Figure 6. The code of the first block (lines 1-6) checks if the caller is the owner (represents the code of the modifier `onlyOwner`). If so, the execution is directed to the block at line 8 to execute the code of the function `switchLiquidity`. Otherwise, the execution goes to the block at line 7, where `REVERT` instruction halts the execution, and reverts changes made to the contract state.

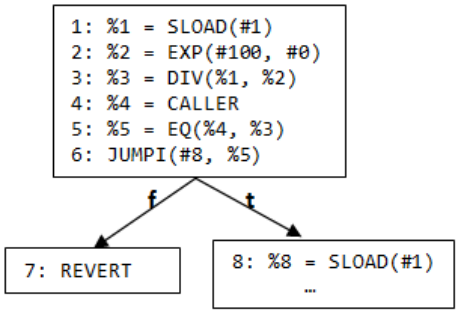

Fig. 6. Bytecode of the function switchLiquidity in Figure 1.

To identify the access control check performed in this function, *AChecker* performs backward data-flow analysis starting at the condition of the `JUMPI` instruction at line 6. *AChecker* will find the data-flow `%1=SLOAD(#1), %2=EXP(#100,#0), %3=DIV(%1,%2), %4=CALLER, %5=EQ(%4,%3)`. The instructions `%1=SLOAD(#1)` and `%4=CALLER` are part of this data-flow to evaluate the condition of the `JUMPI` instruction. Thus, the condition of the `JUMPI` at line 6 is identified as an access control check, and the storage location #1 (read by instruction `%1=SLOAD(#1)`) is identified as an access control state variable (that should not be written by the contract users).

A challenge faced in our analysis is that the conditions obtained by the backward data-flow analysis are (by design) an over-approximation of the checks (conditions) in the contract code that have data dependency on `CALLER` and `SLOAD` instructions. Thus, the filtered checks may contain non-access control checks that behave similar to access control checks (i.e., reference the contract's caller, and load a value from storage). These cases result from conditions in which the contract caller is used as a key to access values in storage mappings, such as $require(balances[msg.sender] >= amount)$.

To address this challenge, *AChecker* excludes these conditions from the set of conditions identified as access control checks. The core idea is to use data-flow analysis to differentiate between the mappings representing access control state variables and other mappings. The idea is based on two observations. First, the values of mappings used as access control state variables are only used for validating callers, not for any other operations, e.g., mathematical operations.

Second, mappings representing access control state variables usually store Boolean values. Based on these two observations, *AChecker* performs further analysis of the conditions returned by the backward data-flow analysis in which the contract caller is used to access mappings (lines 6-14 in Algorithm 1).

For each condition returned by the backward data-flow analysis, *AChecker* confirms the condition as an access control check if the corresponding state variable used in the condition is not a mapping (lines 7-8). Otherwise, if the state variable is a mapping and stores Boolean values, then *AChecker* confirms the condition as an access control check (lines 9-10). Although type information is not available at the EVM bytecode level, we found that the data type of the values stored in mappings can be inferred from the bytecode. This is because for mappings storing Boolean values, the compiler uses AND bitmasking with Boolean values, where any Boolean value loaded from the storage gets bitmasked using '0xff' (Boolean holds one byte of the storage location). Finally, if the state variable is a mapping and does not store Boolean values, *AChecker* performs static forward data-flow analysis starting at the `SLOAD` instruction to check if the mapping referenced in the condition is used by any other operation following the condition. If no such data-flow is found, the condition gets confirmed as an access control check (lines 11-14). The analysis stops when reaching a `STOP` or `RETURN` instruction.

After identifying access control checks and the corresponding state variables, *AChecker* searches for critical instructions in the contract code and proceeds to the next step.

C. Violated/Missing Access Control Checks Detection

In the second step, *AChecker* performs static taint analysis to check if any of the access control state variables and critical instructions are manipulated by attackers. The performed taint analysis is both inter-procedural and context-sensitive, and tracks taints through the contract storage.

The analysis uses user inputs as sources of taint, and uses critical instructions along with access control state variables as sinks. Table I shows the sinks used by *AChecker* for each vulnerability. Each `SSTORE` instruction that writes to any of the identified access control state variables is used as a sink to detect violated access control checks. Further, `SELFDESTRUCT` instructions, and the address parameters of `SELFDESTRUCT` and `DELEGATECALL` instructions are used as sinks to detect missing access control checks, as the target addresses of `SELFDESTRUCT` and `DELEGATECALL` should not be manipulated by unauthorized users. If the attacker is able to control the address parameter of `SELFDESTRUCT`, they will receive the funds of the contract when the `SELFDESTRUCT` is executed, as discussed in Section II-B. Similarly, setting the address of the `DELEGATECALL` will allow attackers to execute their own code in the context of the contract, thereby manipulating the contract state and changing its behavior.

A sanitizer is a code check that prevents contract users from either controlling a critical instruction or manipulating an access control state variable. There are two types of *unsanitized* taint flows that are considered by *AChecker*, (1) taint flows to access control state variables are considered as *violated access control checks*, and (2) taint-flows to critical instructions are considered as *missing access control checks*. Finding taint flows to access control state variables and critical instructions results in discovering access control violations as well as missing access control, where the attacker is able to manipulate the access control state variables or critical instructions. This includes the cases when incorrect access control modifier names that can be bypassed by attackers are used as access control checks.

TABLE I
EVM INSTRUCTIONS DEFINED AS SINKS BY *AChecker*. THE ARGUMENT IN **BOLD** IS THE SINK; OTHERWISE, THE WHOLE INSTRUCTION IS A SINK.

Vulnerability	EVM Instruction
VAAC	`SSTORE` <key, **value**>, key ∈ AC state variables
MAAC	`SELFDESTRUCT`
MAAC	`SELFDESTRUCT` <**addr**>
MAAC	`DELEGATECALL` <, **addr**, ...>

D. Potentially Intended Behaviors Filtering

One of the challenges faced when analyzing smart contracts for access control vulnerabilities is the intended behavior cases (Section II-C). Based on our observations, we found that a lack of access control does not always result in a vulnerability, as the developer may implement some other non-access control checks. The contract code may *intentionally* allow even unauthorized users of the contract (any Ethereum account) to manipulate access control state variables but only under specific non-access control constraints. These cases are challenging for static analysis approaches to distinguish from vulnerabilities, as there are often no access control policy specifications. Figure 4 shows an example in which unauthorized users can write to the access control state variable `owner` (line 11) when they pay a minimum of 1000 Wei. Static analysis tools would falsely flag this as a vulnerability.

AChecker addresses this challenge by performing symbolic-based analysis to infer these non-access control checks implemented in these cases, and then flagging these cases as potentially intended behaviors. Our intuition in reasoning about intended behavior cases is that an unprotected statement that manipulates an access control state variable, *but only* under specific non-access control constraints, is likely to be a potentially intended behavior implemented by the developer.

Definition 4 (Potentially Intended Behavior)**.** *In a smart contract, manipulating an access control state variable with no access control check, and under other non-access control constraints, is most likely an intended behavior rather than an access control vulnerability.*

A naive approach to filter potentially intended behaviors would be to flag as potentially intended *any unprotected statement* that updates access control data and is guarded by non-access control checks. However, blindly filtering cases in this way will result in several vulnerabilities falsely reported as intended behaviors. For example, Figure 7 shows a vulnerable function `initialize` implemented to initialize

the access control state variable `operator` (line 4) while `initialized` is 'false'. The developer mistakenly did not add a statement to set the `initialized` to 'true' after line 4; which resulted in a vulnerability. However, deciding intended behavior just based on searching for implemented checks will report this vulnerability as an intended behavior.

```
1 bool public initialized = false ;
2 function initialize () {
3    require (! initialized);
4    operator = msg.sender;
5 }
```

Fig. 7. Example of a vulnerability due to an always True check.

Therefore, to analyze for potentially intended behaviors, *AChecker* uses symbolic execution [26] to infer constraints of manipulating access control state variables, and differentiate potentially intended behaviors from vulnerabilities as below. Our design choice to use symbolic execution is supported by the observation that constraints implemented in intended behaviors are usually few, therefore tractable, and can hence be solved using satisfiability modulo theories (SMT) solvers.

When a taint-flow is found to an access control state variable, *AChecker* executes the taint-flow path symbolically to generate constraints under which the taint flows from the taint source to the access control state variable, then uses an SMT solver to check if the path is feasible. If the path is feasible, *AChecker* uses the SMT solver to find an assignment that makes these constraints unsatisfiable through negating the constraints. If an assignment is found to make the negated constraints satisfiable, then *AChecker* reports this as a potentially intended behavior as the taint-flow is not always feasible – otherwise, *AChecker* reports it as a vulnerability.

For the example in Figure 4, *AChecker* finds that `owner` gets tainted at line 10. Then, *AChecker* symbolically executes the taint-flow path to synthesize the constraints under which `owner` gets tainted. In this case, the synthesized path constraints are "$msg.value \geq howMuchToBecomeOwner \wedge howMuchToBecomeOwner = 1000$". The path is feasible and the SMT solver will find an assignment for `msg.value` that satisfies the negated constraint "$!msg.value \geq howMuchToBecomeOwner \wedge howMuchToBecomeOwner = 1000$", as we are interested in proving that the original constraints (without negation) are not always satisfied. *AChecker* will flag this as a potentially intended behavior as it was able to find cases under which manipulating the state variable `owner` is not always feasible. This is because restricting execution of the code updating `owner` state variable by implementing specific constraints is likely intended by the developer to manage `owner` writes.

When a solution is not found to make the constraints unsatisfiable, if the constraints depend on storage variables getting updated in the taint-flow path, *AChecker* calls the SMT solver again using the updated state. We need this to infer the intended behavior cases usually used for initializing storage variables only once and then set a specific storage variable to prevent future initialization. Further, our symbolic analysis abstracts loop analysis where loops get unrolled to at most three iterations to reduce overhead and filter more potentially intended behavior cases. With that said, we observed from our experiments that loops rarely get used in paths that manipulate access control state variables.

E. Implementation

The implementation of *AChecker* relies on *eTainter* [18], a static taint analyzer for smart contracts, to perform taint analysis, and on *teEther* [27], a symbolic analysis framework for smart contracts, to generate the control flow graph (CFG) and perform symbolic analysis. We built up on the symbolic-execution module of *teEther* to perform our intended analysis, specifically, to unroll loops and negate generated constraints when inferring potential intended behavior.

IV. Evaluation

We evaluate *AChecker* by answering the following three research questions (RQs):

RQ1. How effective is *AChecker* in detecting access control vulnerabilities compared to the prior work?

RQ2. How precise is *AChecker* in inferring potentially intended behaviors in smart contracts?

RQ3. What is the performance of *AChecker*?

RQ4. How prevalent are access control vulnerabilities in smart contracts?

A. Experimental Setup

1) Datasets: In our evaluation, we use three datasets, as summarized in Table II. The first is the *CVE dataset* consisting of 15 contracts, where each contract is assigned a CVE (Common Vulnerability and Explosures) for an access control vulnerability. This dataset has been collected by the authors of SPCon [8] to evaluate their tool and compare with the prior work. They originally collected 17 contracts in the SPCon paper, but they reported that one contract is not the vulnerable contract discussed by the CVE-2021-3006 (the disclosing report of this CVE referenced a non-vulnerable contract). Another contract (CVE-2018-17111) has a vulnerability in a modifier that is not used by any function in the contract, so the code of the modifier is unreachable. Therefore, we omitted these two contracts from the dataset.

The second dataset is the *SmartBugs-wild* public dataset which consists of about 47,518 contracts. The SmartBugs-wild dataset has been collected by Durieux et al. [17] to evaluate nine static analysis tools. Finally, the third dataset is the *Popular-contracts dataset* from eTainter [18]. This dataset consists of 3,000 contracts deployed on the Ethereum blockchain, which have the largest number of transactions as of January 2022. Thus, this dataset represents contracts that are highly used in the Ethereum blockchain.

2) Setup: We run the experiments on five Intel Xeon 2.5 GHz machines; each machine has one core. On each machine, we allocate 48GB of RAM for each run. For all tools, we set a timeout value of 10 minutes (600 seconds) per smart contract.

TABLE II
DATASETS USED IN OUR EVALUATION.

Dataset	Number of Contracts
CVE dataset	15
SmartBugs-wild dataset	47,518
Popular-contracts dataset	3,000

3) Method and Metrics: To answer **RQ1**, we compare the effectiveness of *AChecker* by comparing it to eight analysis tools that target detecting all or a subset of access control vulnerabilities. These tools are SPCon [8], Ethainter [7], Securify [9], Mythril [12], Manticore [10], SmartCheck [14], Maian [11], and Slither [13]. We use two different datasets to compare against prior work; both datasets have been used by previous studies [8], [17] to evaluate the tools we are comparing with. First, we use the CVE dataset, in which the vulnerabilities and their locations are known for each contract. Thus, we have the ground truth for our comparison. For each tool, we run the tool on the contracts in the dataset and then count how many vulnerabilities the tool detects. This allows us to estimate the *recall* of each tool. As Ethainter's source code is not publicly available, in our experiments, we use the online deployment of Ethainter [28] mentioned in their paper.

Second, as the CVE dataset has limited contracts, we use the SmartBugs-wild dataset to compare precision of the tools. SmartBugs-wild dataset has no ground truth of the vulnerabilities in each contract. We thus follow the approach of SPCon [8] to manually inspect a subset of vulnerabilities flagged by each tool in order to determine whether these vulnerabilities are true-positives or false-positives. For the tools that detect other classes of vulnerabilities, only the access control vulnerabilities are selected for inspection and for calculating the precision (which is a fraction of true access control vulnerabilities of all selected samples). This does not have an effect on the precision of these tools as the reported access control false-positives by these tools are not true vulnerabilities of other types detected by the tools.

To avoid bias, we focused on the subset of vulnerabilities that was selected by the authors of SPCon, and borrowed their inspection results for the tools we used for comparison. In that work, for each tool that reported a low number of vulnerabilities (Maian, Manticore, and SPCon), all the reported cases were inspected (44, 47, and 44, respectively). For all other tools, statistical sampling was used to obtain a 90% confidence level and a margin of error of 10% on whether the sample is representative of all reported cases. Thus, the number of samples selected is different for each tool. We applied the same approach to select samples for *AChecker*: approximately 60 samples. Further, for *AChecker*, two smart contract security researchers independently inspected the vulnerabilities reported by the tool and only the vulnerabilities agreed on by both researchers are considered as true vulnerabilities. We use the inspection results to estimate the *precision* of each tool.

To answer **RQ2**, we run *AChecker* on the SmartBugs-wild dataset and randomly select a subset of 10 contracts marked by *AChecker* as having *potentially intended behaviors*; we add to this set 10 randomly selected contracts marked by *AChecker* as vulnerable. We recruited a third-party expert – a graduate student with smart contracts security auditing expertise, who is not an author of this paper. We asked the expert to manually inspect the contracts to determine which constitute true vulnerabilities, without knowing about our classification of intended behaviors. Our goal is to check if the results of the expert's classification of the intended behaviors match the results reported by *AChecker*. We also check whether the contracts we mark as potentially intended behaviors are also reported as vulnerabilities by prior work.

To avoid bias, we used the reported analysis results for Securify, Mythril, Manticore, SmartCheck, Maian, and Slither, and ran the other two tools, Ethainter and SPCon on these contracts ourselves (as prior work did not report these results).

To answer **RQ3**, we run *AChecker* on SmartBugs-wild and Popular-contracts datasets, and calculate the average analysis time of the contracts that *AChecker* analyzed successfully. When *AChecker* cannot analyze a contract within the timeout-value window, we consider *AChecker* as unable to analyze it.

To answer **RQ4**, we count contracts flagged as having access control vulnerabilities by *AChecker* on the SmartBugs-wild dataset and Popular-contracts dataset.

B. Experimental Results

RQ1 (Effectiveness):

a) **CVE dataset:** Table III shows the results of comparing *AChecker* with the prior work on the CVE dataset. In the table, ✓denotes when the tool was able to detect the vulnerability for the corresponding CVE. The bottom row shows the overall recall (detection rate) for each tool on the CVE dataset.

From the table, we see that AChecker *detected most of the vulnerabilities, and has a higher recall than all the other tools.* In particular, *AChecker* detected eight of the nine vulnerabilities detected by the other tools, and *four additional vulnerabilities* detected by none of the other tools (Section V discusses the reason for the three undetected vulnerabilities). Thus, *AChecker* has an overall **recall** of **80%** on the CVE dataset. SPCon had the next best recall (60%). The other tools all had less than 30% recall on the CVE dataset.

b) **SmartBugs-wild dataset:** Table IV shows a summary of the results of comparing *AChecker* with the other existing tools on the SmartBugs-wild dataset. The first row (Reported) shows the total number of vulnerabilities reported by the tool, the second row (Inspected) shows the number of sample vulnerabilities we manually inspected to estimate the precision of the tool, and finally the row at the bottom (Precision) shows the estimated precision of the tool.

From Table IV, the results show that *our proposed approach,* AChecker*, has the highest* **precision (88.3%)** *among all the tools.* Further, the tools with the second best precision are SPCon (81.8%) and Maian (61.4%). However, we see from the table that SPCon and Maian reported a much smaller number of vulnerabilities, with 44 vulnerabilities reported by each tool in comparison to 624 vulnerabilities reported by

TABLE III
COMPARISON WITH PRIOR WORK ON THE CVE DATASET (RECALL).

CVE	Slither	Maian	SmartCheck	Manticore	Mythril	Securify2	Ethainter	SPCon	*AChecker*
CVE-2018-10666								✓	✓
CVE-2018-10705								✓	✓
CVE-2018-11329								✓	✓
CVE-2018-19830									✓
CVE-2018-19831					✓			✓	✓
CVE-2018-19832		✓			✓			✓	✓
CVE-2018-19833									✓
CVE-2018-19834									✓
CVE-2019-15078		✓			✓			✓	✓
CVE-2019-15079								✓	
CVE-2019-15080								✓	✓
CVE-2020-17753	✓		✓		✓				
CVE-2020-35962									
CVE-2021-34272								✓	✓
CVE-2021-34273									✓
Recall%	6	13	6	0	26	0	0	60	80

TABLE IV
COMPARISON WITH PRIOR WORK ON THE SMARTBUGS-WILD DATASET (PRECISION).

	Slither	Maian	SmartCheck	Manticore	Mythril	Securify	SPCon	*AChecker*
Reported	2,356	44	384	47	1,076	614	44	624
Inspected	66	44	58	47	64	61	44	60
Precision%	24	61	29	19	39	33	81.8	88.3

AChecker. This could be an indicator that these tools missed a lot of true vulnerabilities, and hence they may have lower recall than what we have seen earlier on the CVE dataset.

To establish a better understanding of the capability of the prior work to detect existing vulnerabilities in the SmartBugs-wild dataset, we checked how many of the true vulnerabilities reported by *AChecker* (we have manually confirmed these as true vulnerabilities), were reported by each of the existing tools. To avoid bias, we have used the original analysis results available with SmartBugs-wild dataset [17] for six tools: Slither, Maian, SmartCheck, Manticore, Mythril, and Securify. We ourselves ran the other two tools (Ethainter and SPCon) that were not considered in the SmartBugs study [17].

TABLE V
COMPARISON WITH PRIOR WORK ON THE SMARTBUGS-WILD DATASET (RECALL).

TPs reported by *AChecker*	Slither	Maian	SmartCheck	Manticore	Mythril	Securify	Ethainter	SPCon
53	11	6	0	0	11	5	0	10

The results are shown in Table V. The results show that *out of the 53 true vulnerabilities that are detected by* AChecker*, all the other tools together detected only 24 vulnerabilities.* We also cross-checked the vulnerabilities reported by other tools from the total true positives. *AChecker* (69.62%) outperformed all tools in reporting true vulnerabilities: Slither (44.4%), Maian (15.8%), Smartcheck (15.8%), Manticore (5.9%), Mythril (43.3%), Securify (15.2%), Ethainer (7%), SPCon (21%).

Answer to RQ1: *AChecker* outperforms all existing tools for detecting access control vulnerabilities, and is able to detect more true vulnerabilities with fewer false alarms.

RQ2 (Potentially Intended Behaviors): Table VI shows the results of the manual inspection for the 20 selected contracts. The first row shows the results for contracts that are marked by *AChecker* as vulnerable and the second row shows the results for contracts that are marked as potentially intended behaviors. The results demonstrate that the third-party expert classified eight of the 10 vulnerable cases reported by *AChecker* as true vulnerabilities, and two as false-positives. One false-positive is for a code protected by an access control checked in another external contract; hence, not recognized by *AChecker*. In the second case, the contract allows anyone to update the contract manager, but it calls an external code to authenticate the new manager, and it may eventually store the authenticated manager – the external code is not available to verify, though.

Further, the third-party expert classified *all the 10 potentially intended behavior cases* reported by *AChecker* as benign cases intentionally implemented rather than vulnerabilities. Moreover, we found that 70% of the 10 intended behavior cases were reported by the existing tools as vulnerabilities, adding to the false-positives of these tools. Overall, *AChecker* reported 418 functions with intended behavior cases. 221 of these cases are also reported by other tools as having access control vulnerabilities (though we did not validate them). This highlights the need for an approach like *AChecker* that filters the intended behaviors from exploitable true vulnerabilities.

Answer to RQ2: Our approach for filtering potentially intended behavior has high precision in inferring the true intended behavior cases.

TABLE VI
INTENDED BEHAVIOR FILTERING (PRECISION).

Classification	Samples	True Vulnerabilities
Vulnerable	10	8
Potentially intended behavior	10	0

RQ3 (Performance): By running *AChecker* on all the contracts in the SmartBugs-wild dataset, we found that the average analysis time of *AChecker* per contract is 11.74 seconds. This time includes the time taken by the symbolic execution to filter potentially intended behaviors. We obtained a similar average analysis time, 10.55 seconds for *AChecker* when running on the Popular-contracts dataset. In both datasets, *AChecker* timed out in about 17% of the contracts. Most of the timeouts were due to the inability of teEther [7] to generate the CFG.

Answer to RQ3: *AChecker* has an average analysis time of about 11 seconds per contract.

RQ4 (Prevalence of Access Control Vulnerabilities):
SmartBugs-wild dataset: As mentioned in the results of RQ1, *AChecker* flagged 624 contracts as having access control vulnerabilities with a precision of about 88%.
Popular-contracts dataset: Table VII shows the results of analyzing the Popular-contracts dataset by *AChecker*. Results show that *AChecker* flagged 21 contracts as having access control vulnerabilities. To determine how many of these reported vulnerabilities are true vulnerabilities, we inspected all the reported cases in all the contracts whose source code is available on Etherscan [29], an explorer of the Ethereum blockchain. We restricted ourselves to contracts with source code as it is difficult to inspect the bytecode of contracts for vulnerabilities. Further, without the source code, running a smart contract following the trace found by taint-flow and symbolic execution is insufficient to identify vulnerabilities. Out of the flagged 21 contracts, 10 contracts have source code available. Our manual inspection of the vulnerabilities in these 10 contracts show that 9 of the vulnerabilities are true vulnerabilities (90% precision), and one is false-positive.
Answer to RQ4: *AChecker* flagged 624 and 21 vulnerable contracts in the SmartBugs-wild and Popular-contracts datasets, respectively, showing that access control vulnerabilities are prevalent in these datasets.

TABLE VII
RESULTS OF ANALYZING POPULAR-CONTRACTS BY *AChecker*.

Flagged	Inspected	TPs
21	10	9 (90%)

V. DISCUSSION

A. Result Analysis

As mentioned in RQ1, *AChecker* did not report three vulnerabilities in the CVE dataset. In one case, `tx.origin` is used in an access control check instead of `msg.sender`. *AChecker* does not analyze for this case as it is an outdated vulnerability that is found mostly in old contracts and can be found using a string search. Of the existing static analysis tools, Mythril, Slither, and SmartCheck can detect this vulnerability. Adding support for `tx.origin` in *AChecker* is straightforward, and is a potential direction for future work.

The other two cases are for functions that lack access control. These two cases are out of the scope of *AChecker* as *AChecker* analyzes for missing access control for a set of predefined instructions. In general, static analysis approaches such as *AChecker* cannot decide the contract functions that should be protected, due to the lack of access control specifications. Although SPCon can detect these two cases as it uses the transactions history to mine the access control policy of the contract, neither of these two cases is detected by SPCon.

The results of *AChecker* on the three datasets show that most of the reported vulnerabilities are caused due to having *violated access control checks*. Only two existing analysis tools, SPCon and Ethainter, analyze for these vulnerabilities. However, both tools failed to detect many of the vulnerabilities found by *AChecker*. Ethainter's dependency on specific syntactic patterns was the main reason for its undetected cases.

For SPCon, the contracts that can be analyzed are limited due to having a low number of transactions, and it is challenging to increase the number of transactions to improve SPCon. As discussed (Section I), SPCon requires transactions to be performed by only authorized users of the functions, as per the desired contract security policy, and it is difficult to determine whether or not a transaction obeys the security policy when the contract security policy is not known. This is because transactions that do not obey the security policy would result in increased false-negatives and false-positives.

Moreover, SPCon failed to mine access control roles for several vulnerable contracts where each contract has tens of thousands of transactions. An example taken from a vulnerable contract having about 183,533 transactions [30] is shown in Figure 8. This contract has a vulnerability that enables anyone to become an owner of the contract and perform actions such as withdrawing the contract Ether balance. SPCon was not able to mine the access control role for the vulnerable function `SAC` because this function was called by multiple different accounts; hence SPCon's approach considers it as a public function that can be called by anyone, and ignores it.

```
1 function SAC() public {
2    owner = msg.sender;
3    balances[owner] = totalDistributed;
4  }
```

Fig. 8. Violated access control check vulnerability undetected by prior work.

The results of RQ2 also highlighted the capability of *AChecker* to infer many cases of intended behavior reported as vulnerabilities by the existing analysis tools. An example case is shown in Figure 9. This code is taken from a contract implementing a token named Peculium [31], and there are 14,972 transactions on this contract at the time of writing. The design of the contract allows the freeze possibility of tokens. Thus, the `transfer` function (line 8) has an access control check at line 9 so that only unfrozen accounts (stored at `balancesCanSell`) can transfer tokens. The contract allows token owners to defrost their tokens after a predefined date using the function `defrostToken`. Analyzing constraints of the `defrostToken` function enabled *AChecker* to infer that this is a potential intended behavior of the contract.

```
1  mapping(address => bool) public balancesCanSell;
2  uint256 public dateDefrost;
3  function defrostToken() public
4  { // Defrost your tokens, after the date of the defrost
5    require(now>dateDefrost);
6    balancesCanSell[msg.sender]=true;
7  }
8  function transfer(address _to, uint256 _value) public
       returns (bool) {
9    require(balancesCanSell[msg.sender]);
10   return BasicToken.transfer(_to,_value);
11 }
```

Fig. 9. Intended behavior reported as a vulnerability by prior analysis tools.

B. Limitations

AChecker has three main limitations. First, it performs intra-contract analysis and hence does not analyze access control delegated to other contracts. Second, it analyzes missing access control only for a set of predefined critical instructions due to the lack of access control specifications of contracts. Thus, *AChecker* does not analyze missing access control for functions that should be protected if they do not have predefined instructions. Finally, *AChecker* only checks access granted to the contract callers on functions and, what we call, critical instructions. This is a common way to authorize users in smart contracts, but other methods are also possible.

C. Threats to Validity.

An external threat to validity is the limited number of contracts used to evaluate effectiveness of *AChecker* and compare it to the prior work (15 contracts). We partially mitigated this threat by using a dataset with a confirmed set of CVEs. In addition, we used the Smartbugs dataset consisting of 47,518 real-world contracts, and the Popular-contracts dataset (with 3,000 real-world widely used contracts) to evaluate the precision of *AChecker* and compare with the prior work. Both datasets have been used by previous studies to evaluate the analysis tools we use for comparison. We avoided using datasets injected with artificial vulnerabilities as, to our knowledge, there is no existing tool that injects access control vulnerabilities covered by the proposed approach - injecting such vulnerabilities ourselves may bias the evaluation.

An internal threat to validity is the possible bias due to manual inspection of the sample vulnerabilities to compare with the prior work. We partially mitigated this threat in two ways. First, we used the inspection results from previous studies for the analysis tools we are comparing with *AChecker*, where possible. Second, two smart contract researchers independently inspected the vulnerabilities of *AChecker*, and only those agreed on by both are counted as true vulnerabilities.

VI. Related Work

Several static analysis tools have been proposed for detecting various security bugs in smart contracts [32], [27], [11], [33], [10], [34], [9] , [7], [14], [8], [13], in addition to other studies [17], [35] that evaluated the efficacy of these tools. Many of these proposed tools targeted detecting access control vulnerabilities [9], [14], [10], [12], [11], [27], [7], [8]. SPCon [8] analyzes smart contracts for permission bugs. SPCon first uses the contract transactions history available on the blockchain to mine access control roles of the contract. Then it detects permission bugs via conducting symbolic-analysis-based conformance testing. However, the dependency on the transaction history to mine access control roles makes SPCon unable to mine several access control roles; hence it leaves several bugs undetected (as we have shown).

Another work, Ethainter [7], was the first tool to focus on detecting composite vulnerabilities (i.e., access control vulnerabilities). Ethainter statically analyzes smart contracts for composite vulnerabilities by performing information-flow analysis using inference rules of a designed abstract input language of the bytecode. However, Ethainter's approach has several drawbacks. First, Ethainter's inference rules expect contracts to be coded in a specific style; hence the access control checks detected by these rules are rather rigid [36], which results in several false-negatives and false-positives. Second, Ethainter's rules coarsely over-approximate defining access control checks, thereby defining several irrelevant conditions as access control checks, resulting in high false-positive rate.

Several tools use static analysis to detect general security bugs in smart contracts, including cases of access control vulnerabilities. Securify [9] checks for the presence or lack of specific code patterns of access control before some code statements, such as those that kill the contracts, through checking for compliance or violation of pre-defined security patterns. Maian [11] only analyzes for suicidal and prodigal contracts, in which attackers can kill or steal Ether. Smartcheck [14] is an AST-based approach that employs XPath pattern matching to detect several bugs. However, only a few cases of access control vulnerabilities can be expressed as XPath patterns. Slither [13] performs taint-analysis in the contract source code to detect several bugs. However, Slither supports the detection of a few cases of access control vulnerabilities.

Other tools use symbolic execution to find vulnerabilities. Mythril [12] uses symbolic execution to detect a large group of security bugs, including cases of access control bugs. However, the need to execute multiple functions in sequence to reach the vulnerable state and detect vulnerabilities results in false-negatives. Manticore [11] is another symbolic analysis tool for smart contracts. However, similar to Mythril, its detection capability is limited by the number of transactions needed to trigger bugs, and it only looks for arbitrary code execution through `delegatecall`. Finally, teEther [27] employs symbolic execution to generate exploits for a group of access control-similar vulnerabilities; however, combining symbolic paths of several functions to generate exploits makes teEther scales to only a fraction of Ethereum smart contracts [7].

VII. Conclusion

Access control vulnerabilities in smart contracts can lead to significant financial loss. This paper proposed *AChecker*, a static analysis approach for finding violated and missed access control checks in smart contracts. *AChecker* employs a data-flow analysis technique to determine access control checks implemented in the contract, and a symbolic execution-based approach to distinguish cases implemented as intended behavior from vulnerabilities. We evaluated *AChecker* on three smart contract datasets, and the results show that *AChecker* effectively discovered access control vulnerabilities with a much higher recall and precision than eight existing analysis tools. Further, *AChecker* flagged 21 contracts of the most popular Ethereum contracts as having access control vulnerabilities - 90% of these were true vulnerabilities. Finally, *AChecker* has an average analysis time of about 11 seconds per contract.

ACKNOWLEDGMENT

This work was supported in part by a grant from the Natural Sciences and Engineering Research Council of Canada (NSERC), and Four-Year Fellowship from UBC. We thank the anonymous reviewers of ICSE'23 for their helpful comments.

REFERENCES

[1] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[2] (2017) Cardano blockchain. [Online]. Available: https://www.cardano.org/en/what-is-cardano

[3] Solidity. [Online]. Available: https://docs.soliditylang.org/en/v0.5.1

[4] Ethereum networks. [Online]. Available: https://ethereum.org/en/developers/docs/networks

[5] (2021) Value defi-rekt 2. [Online]. Available: https://rekt.news/value-rekt2

[6] (2018) Accidental's bug froze $280 million worth of ether in parity wallet. [Online]. Available: https://www.cnbc.com/2017/11/08/accidental-bug-may-have-frozen-280-worth-of-ether-on-parity-wallet.html

[7] L. Brent, N. Grech, S. Lagouvardos, B. Scholz, and Y. Smaragdakis, "Ethainter: a smart contract security analyzer for composite vulnerabilities," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 454–469.

[8] Y. Liu, Y. Li, S.-W. Lin, and C. Artho, "Finding permission bugs in smart contracts with role mining," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 716–727.

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

[10] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1186–1189.

[11] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proceedings of the 34th annual computer security applications conference*, 2018, pp. 653–663.

[12] (2019) Mythril. [Online]. Available: https://github.com/ConsenSys/mythril

[13] J. Feist, G. Grieco, and A. 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)*. IEEE, 2019, pp. 8–15.

[14] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2018, pp. 9–16.

[15] M. S. Hecht, *Flow analysis of computer programs*. Elsevier Science Inc., 1977.

[16] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, 2003.

[17] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 ethereum smart contracts," in *Proceedings of the ACM/IEEE 42nd International conference on software engineering*, 2020, pp. 530–541.

[18] A. Ghaleb, J. Rubin, and K. Pattabiraman, "eTainter: Detecting gas-related vulnerabilities in smart contracts," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, p. 728–739.

[19] (2022) AChecker. [Online]. Available: https://github.com/DependableSystemsLab/AChecker

[20] Business alliance financial circle (BAFC) token. [Online]. Available: https://etherscan.io/address/0x9924A7E3A2756Ab8B9A828485f052b6693AaA33E

[21] Unprotected selfdestruct instruction. [Online]. Available: https://swcregistry.io/docs/SWC-106

[22] Unprotected ether withdrawal. [Online]. Available: https://swcregistry.io/docs/SWC-105

[23] Delegatecall to untrusted callee. [Online]. Available: https://swcregistry.io/docs/SWC-112

[24] Airdrop contract. [Online]. Available: https://etherscan.io/address/0x220348263aab5a038845483f6096895aa59f3977

[25] Sugoi nft nyc 2022. [Online]. Available: https://etherscan.io/address/0x8088f4612eadb9d60d5c8abf4a9d0fdfc3df2f1e

[26] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[27] J. Krupp and C. Rossow, "teEther: Gnawing at ethereum to automatically exploit smart contracts," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 1317–1333.

[28] Web. (2021) contract-library. [Online]. Available: https://contract-library.com

[29] Etherscan. https://etherscan.io.

[30] Sachio (sch) contract. [Online]. Available: https://etherscan.io/address/0xf34839b310097fcb4cf3a302dda8cc9b57501083

[31] Peculium (pcl) token. [Online]. Available: https://etherscan.io/address/0x53148bb4551707edf51a1e8d7a93698d18931225

[32] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 254–269.

[33] E. Zhou, S. Hua, B. Pi, J. Sun, Y. Nomura, K. Yamashita, and H. Kurihara, "Security assurance for smart contract," in *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. IEEE, 2018, pp. 1–5.

[34] C. F. Torres, J. Schütte, and R. State, "Osiris: Hunting for integer bugs in ethereum smart contracts," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 664–676.

[35] A. Ghaleb and K. Pattabiraman, "How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 415–427.

[36] Y. Smaragdakis, N. Grech, S. Lagouvardos, K. Triantafyllou, and I. Tsatiris, "Symbolic value-flow static analysis: deep, precise, complete modeling of ethereum smart contracts." *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, pp. 1–30, 2021.