Panda: Security Analysis of Algorand Smart Contracts

Zhiyuan Sun[1,2], Xiapu Luo[1*], Yinqian Zhang[2*]

[1]The Hong Kong Polytechnic University
[2]Southern University of Science and Technology

Abstract

Algorand has recently grown rapidly as a representative of the new generation of pure-proof-of-stake (PPoS) blockchains. At the same time, Algorand has also attracted more and more users to use it as a trading platform for non-fungible tokens. However, similar to traditional programs, the incorrect way of programming will lead to critical security vulnerabilities in Algorand smart contracts. In this paper, we first analyze the semantics of Algorand smart contracts and find 9 types of generic vulnerabilities. Next, we propose Panda, the first extensible static analysis framework that can automatically detect such vulnerabilities in Algorand smart contracts, and formally define the vulnerability detection rules. We also construct the first benchmark dataset to evaluate Panda. Finally, we used Panda to conduct a vulnerability assessment on all smart contracts on the Algorand blockchain and found 80,515 (10.38%) vulnerable smart signatures and 150,676 (27.73%) vulnerable applications. Of the vulnerable applications, 4,008 (4.04%) are still on the blockchain and have not been deleted. In the disclosure process, the vulnerabilities found by Panda have been acknowledged by many projects, including some critical blockchain infrastructures such as the decentralized exchange and the NFT auction platform.

1 Introduction

As decentralized cryptocurrencies continue to evolve, the problems of the old blockchain platforms (e.g., Bitcoin [1] and Ethereum [2]) gradually emerge, and new ones are being developed. Traditional blockchain platforms face many problems, such as heavy electricity consumption and high latency due to the Proof-of-Work consensus protocol [3,4]. Solving the problems left by the old platforms has become the driving force for the development of new platforms.

Algorand is proposed to overcome the blockchain trilemma, or the three fundamental difficulties that blockchain system faces today: security, scalability, and decentralization, by adopting a new consensus protocol [5]. It is the first blockchain that provides immediate transaction finality, and its transaction throughput is comparable to large payment and financial networks since its blocks can be finalized in seconds. To promote the development of Algorand, the Algorand Foundation proposed a 250 million Algo Grants Program in 2020 [6]. As a result, Algorand has grown rapidly in recent years, becoming one of the most popular blockchain platforms. In March 2022, the number of transactions on the Algorand platform exceeded 11 million a week [7], and the total number of accounts exceeded 23 million [8].

In contrast to traditional contract law, smart contracts enable traceable and irreversible transactions without the need for third parties. However, smart contracts bring new security challenges, and attackers are particularly motivated to uncover and exploit vulnerabilities in smart contracts that hold cryptocurrencies due to their considerable monetary value. For example, the attack on the DAO contract resulted in a loss of $60 million for the Ethereum community [9].

Unlike Ethereum and other blockchain platforms with only one type of smart contract, Algorand supports two kinds of smart contracts. One is stateful smart contracts, which implement the business logic of applications, similar to the smart contracts in Ethereum and other blockchain platforms. The other one is stateless smart contracts (aka smart signatures), which are attached to transactions and used to determine whether or not the transactions should be approved. By doing so, Algorand can implement some more complex logic and functionality such as delegate signature authority (see §2.3.2). To avoid ambiguity, in this paper, we use applications and smart signatures to represent stateful smart contracts and stateless smart contracts, respectively, and use smart contracts to cover both stateful smart contracts and stateless smart contracts. To facilitate the development of smart contract based applications, Algorand introduces many new features such as different types of transactions (see §2.4), atomic transfers [10] and Algorand Standard Assets [11]. Moreover, to support these new features, Algorand provides a new runtime for executing its smart contracts.

*The corresponding authors.

Unfortunately, these new features in Algorand enlarge the attack surface of its smart contracts, which could be exploited by adversaries to launch various attacks causing severe financial loss. For example, the attack against the Tinyman contract on the Algorand platform caused a loss of $1.8 million [12]. Even worse, little is known about the potential security weaknesses of Algorand smart contracts and the prevalence of vulnerable smart contracts.

In this paper, we conduct the first systematic study on the security of Algorand smart contracts through three steps. First, we carefully study the design of Algorand with a focus on its smart contracts and identify 9 types of generic vulnerabilities that may exist in Algorand smart contracts. Second, we design a new extensible framework based on symbolic execution to detect vulnerable Algorand smart contracts after tackling several new challenges resulting from Algorand virtual machine (AVM)(see §4.4) and its data types as well as the interactions of two kinds of smart contracts (see §4.3). We formally define the detection rules to capture the 9 types of vulnerabilities (see §5). Third, we develop a prototype of the detection framework named `Panda`. It not only includes the detection methods for uncovering the 9 types of vulnerabilities but also supports new detection methods for revealing unknown vulnerabilities as plugins. Applying `Panda` to 543,412 applications and 775,848 smart signatures, we found that 27.75% applications and 10.38% smart signatures contain at least one of the vulnerabilities. In the disclosure process, the vulnerabilities found by `Panda` have been confirmed in a number of projects, including some critical blockchain infrastructures such as the decentralized exchange (FXDX [13]) and the NFT auction platform (ALGOxNFT [14]).

In summary, we make the following major contributions.

- **Vulnerability discovery.** To the best knowledge, we are the first to conduct a systematic investigation on the vulnerabilities in Algorand smart contracts. After in-depth research, we discovered 9 types of generic vulnerabilities in them.
- **New framework and prototype.** We present `Panda`, the first extensible static analysis framework for detecting vulnerabilities in Algorand smart contracts. The evaluation results show that `Panda` achieves excellent performance. To foster the security research of Algorand smart contracts, we will make `Panda` available to the research community after the vulnerability disclosure is complete. Besides, we construct the first dataset of vulnerable Algorand smart contracts for evaluating `Panda` and future detection tools.
- **Precise detection rules.** After scrutinizing the semantics of the TEAL programs and the 9 types of vulnerabilities, we precisely define the formal detection rules for them. These detection rules strictly follow the TEAL semantics and provide a guarantee of soundness.
- **Comprehensive evaluation.** We use `Panda` to vet the security of 543,412 applications and 775,848 smart signatures. The experimental results show the prevalence of security issues in Algorand smart contracts. Specifically, there are 150,676 (27.73%) applications and 80,515 (10.38%) smart signatures that may have been exposed to the threats posed by these vulnerabilities. In addition, we also investigated possible attacks against vulnerable smart contracts (see §6.4).

2 Background

2.1 Account Management

Algorand adopts the account-based model and stores specific on-chain data in accounts [15], such as Algo balances, asset balances, and the local state of joined applications. Algorand accounts are classified into three categories.

External accounts They have the same definition as the EOA (externally owned accounts) [16] in Ethereum. Each external account is associated with a public key, which can be transformed into an Algorand address, and a private key, which is used to sign transactions.

Application accounts They are controlled by stateful smart contracts (see §2.3.1), each of which is associated with an Algorand address and an application ID. Thus, spending on application accounts depends on the logic of the application instead of private keys like external accounts. Application accounts are similar to Ethereum's contract accounts [16].

Signature accounts Every smart signature (see §2.3.2) can be hashed to obtain a unique Algorand address that represents a signature account [17]. Anyone can submit transactions that spend from a signature account as long as the logic of the smart signature approves it. That is, to spend from a signature account, we only need to create a transaction that makes the logic of the smart signature evaluate to true.

2.2 Algorand Virtual Machine

The Algorand virtual machine (AVM) [18] executes the bytecode compiled from the TEAL programs [19]. In AVM, all the operands are pushed and popped from the stack, and the temporary data is stored in the scratch space. Algorand has two persistent storage, global state and local state, which are stored in the form of key-value pairs. The global state stores the general data of the application, while the local state saves the private data for a specific account. Moreover, there are two data types in AVM: byte strings and unsigned 64-bit integers. When an Algorand application writes data to the blockchain, the blockchain stores both the value and its data type.

There are some commonly used opcodes. Specifically, `bz` and `bnz` are jump opcodes that pop an element from the top of the stack and decide whether to jump to the target label based on its value. The `gtxn` opcode is used to access a specific transaction in a group of transactions, and the `txn` opcode is used to access the transaction currently being processed. Some other opcodes and transaction parameter related content

will be used in the following, and readers may refer to the full specification of Algorand opcodes [20].

2.3 Algorand Smart Contracts

There are two types of smart contracts in Algorand, the stateful smart contracts [21] and the stateless smart contracts (aka smart signatures) [22]. The stateful smart contracts represent applications that reside on the blockchain and are remotely callable. A smart signature is submitted to the blockchain along with a transaction, and its logic is used to determine whether or not the transaction will be approved. Algorand smart contracts are primarily developed in PyTeal [23], and then compiled into the TEAL program.

2.3.1 Stateful Smart Contract

We use a simple example in Listing 1 to explain the concepts of applications. Unlike Ethereum smart contracts, an Algorand application is composed of two programs [24], the approval program (starting on line 4) and the clear state program (starting on line 1). The clear state program is used to handle the `clear-state` transactions, and like most programs, the clear state program in line 2 approves the `clear-state` transaction. The approval program handles the other types of transactions mentioned in §2.4, which is usually more complex. To be specific, the entry point of the approval program starts with a conditional statement in line 13, which is similar to a dispatcher that uses the type of the `OnComplete` parameter [25] to determine which program branch to execute. For example, if we set the `OnComplete` to `UpdateApplication`, then the program will execute the logic in line 7 to update the smart contract. Note that the `OnComplete` is one of the parameters of the application call transactions (see §2.4).

```
1  def clear_state_program():
2      return Approve()
3
4  def approval_program():
5      on_c = Txn.on_completion()
6      appID = Txn.application_id()
7      on_update = Seq([
8          ...
9      ])
10     on_delete = Reject()
11         ... # define on_creation
12         ... # define on_call
13     program = Cond(
14         [appID == Int(0), on_creation],
15         [on_c == OnComplete.NoOp, on_call],
16         [on_c == OnComplete.UpdateApplication, on_update],
17         [on_c == OnComplete.DeleteApplication, on_delete],
18         ...
19         # handle OptIn and CloseOut
20     )
21     return program
```

Listing 1: An example of application

```
1  def smart_signature():
2      params_conds = And(
3          Txn.type_enum() == TxnType.Payment,
4          Txn.fee() == Int(1000),
5          Txn.receiver() == Addr(" ... "),
6          Txn.amount() == Int(10000)
7      )
8      safety_conds = And(
9          Txn.close_remainder_to() == Global.zero_address(),
10         Txn.rekey_to() == Global.zero_address()
11     )
12     recurring_conds = And( ... )
13     program = And(params_conds, safety_conds,
       ↪  recurring_conds)
14     return program
```

Listing 2: An example of smart signature

2.3.2 Smart Signature

Smart signatures refer to stateless smart contracts. The reason why they are called stateless smart contracts is that the global state, local state as well as inner transactions are not allowed to be used in them. Specifically, a smart signature is accompanied by a transaction, and its logic is used to determine whether the transaction will be approved. Listing 2 shows an example of a smart signature that checks the transaction type, fee, receiver, amount (lines 2-7), and some critical parameters (lines 8-11). If all the conditions are met, then the transaction will be approved. Smart signatures are mainly used as signature accounts (see §2.1) or as delegate signature authority. Readers may refer to the official documentation [26] for more details about use cases. Importantly, sometimes a smart signature will specify a transaction that invokes a specific application, where additional transaction parameter checks will be performed in this application, and in this paper, we refer to this application as a `Validator`.

2.4 Algorand Transaction

Algorand supports six types of transactions [27], each of which contains some subtypes. In this paper, we mainly focus on three commonly used transaction types.

Payment The payment transaction is used to send `Algos`, the native currency of the Algorand blockchain, from one account to another. There are two important optional parameters in payment transactions: `CloseRemainderTo` and `RekeyTo`, which are Algorand addresses. If one of these two parameters is set, the transaction will perform some crucial operations in addition to transferring `Algos`. Specifically, if the `CloseRemainderTo` parameter is set, all the remaining balance of the sender's account will be transferred to the `CloseRemainderTo` account [28]. If the `RekeyTo` parameter is set, then the sender's future transaction must be signed by the `RekeyTo` account's private key [29]. In other words, the `RekeyTo` account will take over the sender's account,

and the private key of the sender's account will not be able to sign subsequent transactions.

Asset Transfer The asset transfer transaction is used to transfer assets from one account to another. Similar to the payment transaction, the asset transfer transaction also has the `RekeyTo` parameter, and the `AssetCloseTo` parameter [30] corresponding to `CloseRemainderTo`.

Application Call There are seven types of application call transactions: `create`, `update`, `delete`, `opt-in`, `close-out`, `clear-state` and `NoOp`. Specifically, the `create`, `update`, `delete` transactions are used to create, update and delete a stateful smart contract respectively. If a stateful smart contract uses the local state, users who want to interact with this contract must first send an `opt-in` transaction to it. The `close-out`, `clear-state` transactions are used to delete the local state of a contract from the sender's balance record. The key parameters in an application call transaction include `AppID`, which specifies the application to call, and `Oncomplete`, which determines the program branch will be executed.

3 Vulnerabilities in Algorand Smart Contracts

By delving into the Algorand platform, we discover 9 types of vulnerabilities and one potential risk. For ease of description, we group the vulnerabilities into five categories, three of which target applications (see §3.2, §3.3 and §3.4) and two of which target smart signatures (see §3.5 and §3.6).

3.1 Threat Model

Since data on the blockchain is publicly accessible, attackers can download and audit the deployed smart contracts and exploit the vulnerability to launch attacks. Specifically, attackers only need an external account to exploit the vulnerabilities listed in this paper without the need of any privileges. That is, an attacker can deliberately construct a transaction or a group transaction to bypass logical checks and launch a successful attack. In the exploit against applications, the victim is an application account, and in the attack against smart signatures, the victim is a signature account or an external account.

3.2 Unexpected Delete and Update Operation

There are 7 types of application call transactions including `update` and `delete` transactions as mentioned in §2.4. If an attacker initiates an application `update` transaction (`OnComplete` equals to `UpdateApplication`) and attaches a malicious application in this transaction, then the program will execute the logic in Listing 1 (line 7) and the current application will be replaced by the malicious one after the transaction is recorded in the blockchain. In contrast, if an attacker attempts to submit an application `delete` transaction (`OnComplete` equals to `DeleteApplication`), then the transaction will be rejected (line 10). Specifically, the `Reject()` function in line 10 is equivalent to the instruction sequence `"int 0; return"`. This transaction will be rejected by the blockchain because the top of the stack element is 0 when the application executes the return instruction.

Note that anyone can send application `update` transactions and application `delete` transactions, and whether the transaction is approved depends on the program logic. For example, the program can only allow the application creator to modify the application by comparing the transaction sender's address and the application creator's address. However, things may not always go well, and bad program logic (e.g., a programming mistake) may allow anyone to delete or update applications. Specifically, some developers may expect the application to reject all application `update` and `delete` transactions, but some programming mistakes may cause the program to not perform as expected logic, allowing an attacker to take advantage of the situation. In §B.1, we give a case study about an on-chain application with this vulnerability. When an attacker finds this type of vulnerability, they can perform a denial of service attack by deleting the application or taking over the application account by updating the application code.

3.3 Local State Dependency

As mentioned in 2.3.1, a stateful smart contract is composed of an approval program and a clear state program. When a `clear-state` transaction is submitted, the clear state program will execute. Regardless of whether the program is successfully executed or not, the sender's local state of that application will be deleted permanently [24]. That is, a `clear-state` transaction can force deletion of the sender's local state. Thus, if the program logic depends on other users' local state, it is vulnerable because other users (attackers) can submit a `clear-state` transaction to delete their local states to make the program do some unexpected behavior.

The discovery of this vulnerability was inspired by the forced-ether-to-contract vulnerability in Ethereum [31]. Specifically, some of the Ethereum smart contract developers may incorrectly assume that the fallback or payable functions are executed every time `Ether` is transferred to the smart contract. In fact, when an application that is self-destructing sends its remaining `Ether` to other applications, the recipient's fallback or payable function is not executed. As a result, attackers can use the self-destruct mechanism to break the above assumptions and trigger potential logic flaws. Similarly, some Algorand developers may think that deleting the local state must satisfy the application's specific logic for handling `close-out` and `clear-state` transactions. Unfortunately, this is not always the case, and the `clear-state` transaction will force the deletion of the sending account's local state in the target application regardless of whether the transaction is successfully executed or not.

3.4 Unchecked Transaction Receiver

In Ethereum, users can transfer money to an application while calling it, and all tasks can be completed in one transaction, but Algorand does not support such a feature. To implement similar functionality in Algorand, the atomic transaction group must be used to bind payment transactions and application call transactions together. These two transactions have no relation, as a result, the sender and receiver can be arbitrary. Unfortunately, developers may make some wrong assumptions, e.g., the recipients of these transactions are all current applications. If a smart contract does not check the transaction receiver of the payment transaction or the asset transfer transaction, an attacker can specify the receiver as himself to break the program logic.

For example, suppose an application implements bank-like functionality and asks the user to use the atomic transaction group when making a deposit in order to transfer money while invoking the application. If the application does not check the receiver of the payment transaction, an attacker can set the receiver as himself when making a deposit using group transactions. After the blockchain approves the group transactions, the attacker's balance recorded in the bank increases, but no money is transferred to the bank. Finally, the attacker only needs to withdraw the balance in the bank to make a profit.

3.5 Unchecked Transaction Fee

On Algorand, the sender of the transaction pays the transaction fees. Fees are calculated based on the size of the transaction and the minimum fee is 1000 `microAlgos` (i.e., 0.001 `Algos`) [32]. A user can also choose to increase fees to give the transaction a higher priority to be accepted by the blockchain. However, this feature may be exploited for launching attacks. Consider the example in §2.3.2, if this smart signature is used as a signature account and does not restrict the transaction fees (i.e., removing line 4 of Listing 2), then anyone can use this account to send a transaction with huge fees, and this transaction will wipe out all of its balance.

3.6 Unchecked Transaction Parameters

Consider the example of in §2.3.2, if the smart signature is used as a signature account and does not check some critical transaction parameters (i.e., removing lines 8-11 of Listing 2), then anyone can exploit these defects to make profits. For example, an adversary can set the `RekeyTo` parameter to his public address, and then he will take over this signature account. If the `CloseRemainderTo` parameter is set, then he will bypass the amount and receiver limits and get all the `Algos` in the signature account. Similarly, if a smart signature is used to approve asset transfer transactions and the `AssetCloseTo` parameter is not checked, an adversary can exploit it to get all the assets belonging to the signature account.

4 Panda

4.1 Overview

Figure 1 depicts the workflow and architecture of `Panda`, which consists of 6 major components. We will first introduce each component and then describe how we tackle two technical issues in §4.3 and §4.4.

User Interface The `User Interface` takes in user command-line parameters, including user-defined settings and the smart contract file or the `AppID` to be checked. `Panda` supports several user-defined settings (e.g., Z3 timeout, global timeout, block search deep, and block access count) to meet different needs. Specifically, Z3 timeout refers to the time limit for solving path constraints, global timeout refers to the time limit for the whole analysis, block search deep refers to the maximum depth of the search algorithm, and block access count refers to the maximum access number for each basic block. The purpose of setting these parameters is to serve as heuristic pruning strategies for users. For example, block access count can limit the depth of loops and function calls.

Blockchain Explorer Given `AppID`, the `Blockchain Explorer` loads the program bytecode and the global state from the blockchain via the Algorand SDK [33]. Then, it disassembles the bytecode into a TEAL program using Algorand SDK, and other components will conduct program analysis on this TEAL program. After that, it will save the global state into two symbolic arrays in `Memory Modeler` according to the data type. In a nutshell, `Blockchain Explorer` is the interface between `Panda` and the Algorand blockchain.

CFG Builder The `CFG builder` takes the TEAL program as input and parses the jump labels to construct the control flow graph. Notably, when a smart signature contains a `Validator`, we need to merge the smart signature and the `Validator` into a new smart signature (see §4.3) before constructing a control flow graph for analysis, because the `Validator` may also contain code for checking transaction parameters. In other words, we can think of a `Validator` as a complement to the smart signature code.

Memory Modeler The `Memory Modeler` contains all the symbolic arrays used by `Symbolic Executor`. After `Blockchain Explorer` reads the global state from the blockchain, the `Memory Modeler` will use the loaded data to initialize the global state related symbolic arrays. Moreover, depending on the type of array selector and stored values, a symbolic array may have several entities. Because of Algorand's language features, about a hundred different symbolic arrays are used during symbolic execution. All these symbolic arrays are stored in `Memory Modeler`, and the corresponding access interfaces are also provided. After integrating all symbolic arrays into a single module, these symbolic arrays can be accessed in a uniform way in `Symbolic Executor`. Such design also aims to achieve low coupling between modules.

Symbolic Executor The `Symbolic Executor` consists of an

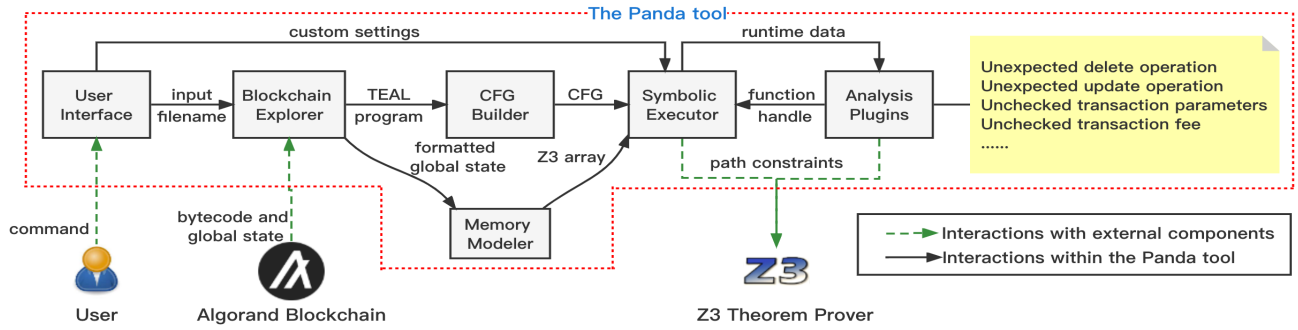

Figure 1: Workflow and architecture of Panda

opcode handle registry and an executor. Specifically, each opcode is implemented as a Python function that is included in the opcode handle registry. This is an extensible programming approach, and each new opcode added in subsequent TEAL versions can be written as a separate Python function and added to this registry so that the Symbolic Executor can support these opcodes. It is worth noting that our tool supports two runtime modes for applications. The symbolic mode treats the data on the blockchain as fully controllable by the user (symbolic variables), while the constant mode treats the data on the blockchain as immutable constant values. When the program is executed, the executor traverses the control flow graph generated by the CFG Builder and executes each instruction symbolically by calling the functions in the opcode handle registry. For each conditional jump instruction, the executor calls Z3 to solve the constraints to prune out unreachable branches. If a return instruction is reached and the top value of the stack is non-zero, a feasible path is found and thus the executor will invoke the Analysis Plugins to detect potential vulnerabilities in the current execution path. During symbolic execution, the Symbolic Executor records a lot of information for vulnerability detection (see §5.1). Note that the data type differences will bring some challenges for the symbolic execution engine (see §4.4).

Analysis Plugins In Analysis Plugins, we provide a registry to which all vulnerability detection rules are added as independent detection functions. This is an extensible programming approach similar to the opcode handle registry in the Symbolic Executor. During the symbolic execution process, once a feasible path is found, the Symbolic Executor will traverse and call all functions in the registry to perform vulnerability detection. Besides, the Symbolic Executor will provide Analysis Plugins with all data structures used to perform vulnerability detection (i.e., all mathematical sets defined in §5.1).

4.2 Symbolic Execution Process

Algorand provides an official opcode specification [20], and Panda strictly follows it. We provide a detailed example in §A to show the symbolic execution process and how the detection rules described in §5 are applied. In the symbolic execution process, Panda converts the termination conditions specified in the specification (such as integer overflow, division by zero, substring opcode out-of-bounds access, etc.) into path constraints. When the path constraints cannot be satisfied, the termination condition is always true, and Panda will treat this as an unreachable path and prune it out.

Since AVM is a stack-based virtual machine, during the symbolic execution process, variables are popped from the stack, and new symbolic variables are constructed and then pushed into the stack again. Every time a variable is pushed onto the stack, Panda will call the built-in function in Z3 to simplify the symbolic variable. When the return instruction is reached, Panda will check whether the symbolic variable at the top of the stack can be a non-zero value under the current path constraints, and if so, the current path is feasible. When a feasible path is found, Panda will perform vulnerability detection on this path. When the symbolic execution process is complete, Panda will output the backtrace of the execution paths of all found vulnerabilities.

Panda models both the global state and the local state as two symbolic arrays, which store two different types of variables respectively. When reading or writing data to the global state or local state, it will determine which symbolic array to operate on according to the data type. Moreover, Panda maintains an instruction cost counter during execution. The maximum number of executed instructions is 700 for applications and 20,000 for smart signatures [34]. When the counter exceeds the limit, the path is considered unreachable, and the executor will backtrack to search for other feasible paths.

For cryptography-related opcodes (e.g., ed25519verify and keccak256), Panda constructs new symbolic variables for representation. To keep the soundness of the evaluation results, Panda currently performs vulnerability detection only when the execution path does not contain cryptography-related opcodes. That is, only execution paths that do not contain cryptography-related opcodes are considered feasible. Note that users can design their own detection rules to allow performing vulnerability detection even if the execution path involves cryptography-related opcodes.

4.3 Handling Smart Signatures with Validators

As shown in §6, the most popular use scenario of smart signature is associating a signature account with a `Validator`. It is relatively easy to detect and verify the vulnerabilities of smart signatures discussed in §3, but things get complicated when the `Validator` is included in smart signatures. Specifically, the checks of transaction parameters in both smart signatures and the `Validator` need to be taken into consideration. To address this challenge, `Panda` merges the smart signature and the `Validator` into a new smart signature before constructing the control flow graph.

Figure 2 illustrates the merging process of a smart signature and the `Validator`. In the first step, the `AppID` of the `Validator` is identified, and its bytecode is downloaded from the blockchain. Next, the jump labels in the smart signature are renamed to prevent the name conflict with the labels in the `Validator`. Afterward, all `return` instructions in the smart signature are replaced with `bnz` instructions that jump to the entry point of `Validator`. By doing so, when the end of the smart signature is reached, `Symbolic Executor` will jump to the entry point of the application and continues execution. Finally, a new smart signature is constructed by directly concatenating the `Validator` to the smart signature. Then, the new smart signature will be further processed to construct the control flow graph.

It is worth mentioning that not all execution paths of a smart signature contain code that calls `Validator`. To handle this issue, when the `Symbolic Executor` executes the instruction "`bnz app_label`", it first checks if the current execution path contains the code that calls `Validator`. If that is the case, it executes the instruction "`bnz app_label`". Otherwise, it treats this `bnz` instruction as the original `return` instruction. That is, if the execution path does not include a `Validator`, the execution should stop at the end of the smart signature instead of jumping to the `Validator` to continue execution.

4.4 Recognizing Data Types

Since Algorand has two different data types, we adopt two new techniques for handling data types.

4.4.1 Runtime Type Checking

Different from other blockchain platforms, there are two data types in Algorand, i.e., the `Uint` type and the `Bytes` type. Furthermore, most of the opcodes in Algorand distinguish the two data types explicitly. For example, the opcode "`+`" pops two `Uint` type variables from the stack and adds them numerically. The `concat` opcode pops two `Bytes` type variables from the stack and concatenates them into a new byte string. If the variable data type does not match the specification of the opcode, the execution will fail and all the operations involved will be reverted. For example, if the `concat` opcode encounters an operand of type `Uint`, the execution will fail.

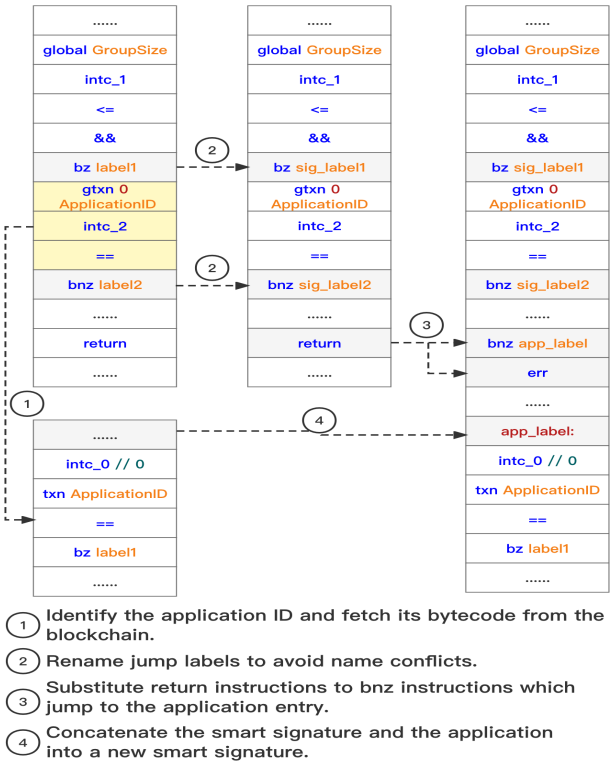

Figure 2: Smart signature merging process

To perform runtime type checking, we use a Python dictionary to represent a variable. Specifically, the dictionary contains two elements: one stores the symbolic value of the variable, and the other stores the data type. By doing so, we can bind the data to its type. During the symbolic execution process, the operand data type of all the opcodes will be verified accordingly, and the computational results are also stored in the Python dictionary. Note that one of the main differences between our tool and other popular symbolic execution tools (e.g., oyente [35] and klee [36]) is that our tool has to handle different data types separately and perform runtime type checking explicitly.

4.4.2 Asynchronous Type Binding

The Algorand storage (i.e., global state and local state) has two data types. The `Bytes` type is internally implemented as a variable-length character sequence while `Uint` type has an 8-byte fixed length. When reading data from the blockchain, the value and its type are loaded together. `Panda` uses Z3 Theorem Prover v4.11.2 [37] as the constraint solver. However, the `Bytes` type variable in Algorand cannot be directly represented by `Bitvector` in Z3, because it is difficult to use Z3 to handle `Bitvector` with dynamic length. Specifically, the length of `Bitvector` is also a symbolic variable, and Z3

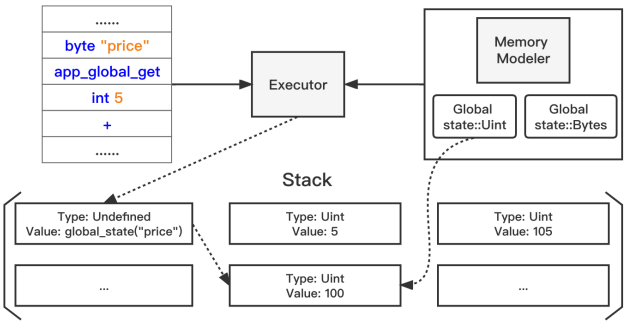

Figure 3: Asynchronous type binding.

currently has no direct way to handle this complex situation. Therefore, we use the Z3 `String` sort and the `Bitvector` sort to simulate `Bytes` type variables and `Uint` type variables. Since Algorand introduces different types of variables, when a symbolic variable is generated (e.g., loading a value from a local state or global state), we should determine its type simultaneously. Incorrect variable types will not pass the runtime type check and will cause the symbolic execution to fail.

To determine the type of the symbolic variables, we propose a new technique named *asynchronous type binding*. Our innovation was inspired by the *lazy binding* technique [38] used in the ELF executable file format. Specifically, lazy binding is the method of deferring symbol resolution for function calls until the first time the function is called. Here we will use a similar approach to determine the variable type at runtime. To be specific, when reading data from the global state of the blockchain, we set the data type to `Undefined`. Only when the executor uses the data later and there is an opcode that can assist in determining the data type, the data type will be finally determined.

Figure 3 illustrates the working process of *asynchronous type binding*. When the executor runs to the `app_global_get` instruction, it pushes an `Undefined` type of value to the stack. The `Undefined` type indicates that the type of this variable is currently unknown and should be determined later. The value field of this variable contains the metadata which will be used to restore the actual value when its type is determined. In this example, the value field indicates that this variable is a global state with the key "price". When the "+" instruction is executed, it instructs the executor to pop two `Uint` type symbolic variables from the stack. Then, the executor will pop two variables from the stack and perform runtime type checking of these two variables. During runtime type checking, the executor finds that the type of one of the variables is `Undefined`. As a result, the executor will define the variable type according to the opcode specification and fetch the symbolic value by parsing the metadata in the value field. In this example, the executor will modify the `Undefined` type variable to `Uint` type and load the symbolic value from the global state with `Uint` type.

In some complex cases, data read from the blockchain may have been copied to multiple storage areas by the `dup` and `store` instructions before the type is determined. To handle this case, we set both the `dup` and `store` instructions to shallow copies, and the new values generated by both instructions point to the same memory area. In this way, when the type of the value in one storage area is determined, other areas will also be determined because the data stored in these areas refers to the same entity.

5 Vulnerability Detection

5.1 Notational Conventions

We now explain some notational conventions to better explain the logic of vulnerability detection.

- `groupTxns` is a set that contains all the accessed transactions by the `gtxn` opcode in the current execution trace.
- `local_accounts` is a set that contains all the accounts involved in the current execution trace.
- `traces` is a set that contains all the distinct opcodes of the current execution trace.
- `GroupSize` represents the atomic transaction group size.
- `Version` represents the TEAL program version.
- `currentTxn` represents the transaction being executed (equivalent to the `Txn` opcode).
- `path_constraints` is a set that contains all path constraints from the program entry point to the end instruction.
- `path_constraint_variables` is a set that contains all the symbolic variables that make up `path_constraints`.

To accurately express the vulnerability detection rules, we define 4 predicates $P(constraints)$, $Q(variables)$, and $R(opcodes)$ and $I(txn,type)$. Specifically, $P(constraints)$ is true if the path constraint set is solvable after adding the new path constraints. In other words, $P(constraints)$ holds if the following set is solvable: $path\_constraints \cup constraints$

$Q(variables)$ holds if none of the variables in the parameter set (i.e. $variables$) are contained in the current path constraint. In other words, $Q(variables)$ is true when there is no constraint on any of the variables in the parameter set. The equivalent form of this predicate is: $variables \cap path\_constraint\_variables = \varnothing$

$R(opcodes)$ holds if at least one opcode in the parameter set is used in the current execution trace. The equivalent form of this predicate is: $opcodes \cap traces \neq \varnothing$

Note that transaction types may not be checked explicitly in Algorand smart contracts. Instead, the transaction type can be uniquely restricted by accessing some transaction fields. For example, if the `Amount` field of a transaction in the atomic transaction group is used in the smart contract (included in the path constraints), then the type of this transaction must be

payment (we call it the implicit transaction type). The predicate $I(txn, type)$ checks the implicit type of the transaction and returns true if the type of the transaction is the same as that specified in the second parameter. Finding the implicit transaction type requires checking a large number of transaction fields. Panda checks all fields strictly according to the official specification [39].

5.2 Detection rules

This section elaborates on the detection rules, whose formal definitions are summarized in Table 1, for the vulnerabilities discussed in §3. These detection rules strictly follow the TEAL semantics and provide a guarantee of soundness but not completeness. For the ease of description, we divide the detection rules into multiple parts and distinguish them with red superscripts. For a better understanding, interested readers may refer to the detection example in §A.

5.2.1 Unchecked Transaction Fee

This type of vulnerability corresponds to detection rule 1 in table 1. In part 1, Panda examines if any code does not check the current transaction fee by checking if it is included in the path constraints. Checking the current transaction fee is equivalent to checking the fee of all transactions in the atomic transaction group. In part 2, Panda checks whether the size of the atomic transaction group is larger than the number of accessed transactions. If so, there is at least one transaction whose parameters are not checked by any code. In part 3, Panda examines whether there is a transaction txn of the atomic transaction group whose transaction fee is not included in the path constraints. In part 4, Panda checks whether the sender of txn can be the address of the signature account. Besides, Panda also checks if the index of txn in the atomic transaction group can be equal to the index of the current transaction. If it is not the case, the current transaction cannot be the vulnerable transaction txn. In part 5, Panda checks whether the senders of all transactions in the atomic transaction group except txn can be arbitrary addresses or zero addresses. This check ensures that the execution path is reachable from any account address.

5.2.2 Unchecked Transaction Parameters

These types of vulnerabilities correspond to detection rules 2-4 in table 1. In rule 2, parts 1, 3, 4, 5, and 6 have similar logic to the four parts in rule 1. In part 1, Panda also checks the TEAL version, as the RekeyTo parameter was introduced in version 2. Parts 2 and 5 also check whether the transaction parameters CloseRemainderTo and AssetCloseTo are equal to ZeroAddress; if one of them is set to a valid address, the RekeyTo parameter will have no effect.

In rule 3, parts 1, 4, 7, and 8 have similar logic to the aforementioned rules. Parts 2 and 5 of this rule check whether the type of the current transaction and the transaction txn is payment because the CloseRemainderTo parameter can only be used in payment transactions. Furthermore, not all smart signatures explicitly check the transaction type. Some smart signatures implicitly determine the transaction type by accessing the parameters of a particular type of transaction. Parts 3 and 6 determine the type of transaction by examining the presence of path constraints on these particular transaction parameters. The detection rule 4 is similar to rule 3, except that parts 3 and 6 of rule 3 check whether the transaction type is payment, while rule 4 checks whether the transaction type is asset transfer.

5.2.3 Unexpected Delete and Update Operation

These types of vulnerabilities correspond to rules 5-6 in table 1. For these vulnerabilities, Panda first checks whether the OnCompletion equals UpdateApplication or DeleteApplication and whether AppID equal to 0. It checks AppID because AppID is 0 only in the create transaction. In other words, the execution path with the constraint AppID equal to 0 is unreachable after the application is created. Next, part 2 ensures that the local state is not used in the execution path. This check is to exclude the special case of indirectly specifying privileged accounts through local state. Finally, Panda checks whether the sender of the current transaction is included in the path constraints. If not, it means any user can update or delete the application.

5.2.4 Local State Dependency

This type of vulnerability corresponds to detection rule 7 in table 1. In part 2, panda checks OnCompletion and AppID. Next, it checks if there exist operations on the local state of other users (not the transaction sender). If so, and the current execution trace contains some transaction-related or state-change opcode (results of part 1), Panda considers the application vulnerable to the local state dependency vulnerability.

5.2.5 Unchecked Transaction Receiver

These types of vulnerabilities correspond to detection rules 8-9 in table 1. In part 1, panda checks whether the current execution trace contains some state-change opcodes. In the bank application example in §3.4, an attack is successful only when the attacker's account balance changes. In part 2, panda checks AppID and GlobalSize. Parts 3 and 4 are used to determine the transaction type. In part 5, panda checks if there is a payment transaction whose Receiver field does not exist in the path constraints or if there is an asset transfer transaction whose AssetReceiver field does not exist in the path constraints. Finally, part 6 ensures that key parameters, such as transaction amount, are used in the execution path.

Index & Mode & Vulnerability Type	Detection Rules
(1) [SIG] Unchecked transaction fee	$^{1}\mathbf{Q}(\{currentTxn.Fee\}) \wedge (^{2}\mathbf{P}(\{GroupSize > \lvert groupTxns\rvert\}) \vee \exists txn \in groupTxns,$ $^{3}\mathbf{Q}(\{txn.Fee\}) \wedge ^{4}\mathbf{P}(\{txn.Sender = LogicAddr, txn.Index = currentTxn.Index\}) \wedge$ $^{5}\bigwedge_{t \in groupTxns - \{txn\}} \mathbf{P}(\{t.Sender = RandomAddr \vee t.Sender = ZeroAddr\}))$
(2) [SIG] Unchecked RekeyTo	$^{1}Version >= 2 \wedge \mathbf{Q}(\{currentTxn.RekeyTo\}) \wedge$ $^{2}\mathbf{P}(\{currentTxn.CloseRemainderTo = ZeroAddr, currentTxn.AssetCloseTo = ZeroAddr\}) \wedge$ $(^{3}\mathbf{P}(\{GroupSize > \lvert groupTxns\rvert\}) \vee \exists txn \in groupTxns, ^{4}\mathbf{Q}(\{txn.RekeyTo\}) \wedge$ $^{5}\mathbf{P}(\{txn.Sender = LogicAddr, txn.CloseRemainderTo = ZeroAddr,$ $txn.AssetCloseTo = ZeroAddr, txn.Index = currentTxn.Index\}) \wedge$ $^{6}\bigwedge_{t \in groupTxns - \{txn\}} \mathbf{P}(\{t.Sender = RandomAddr \vee t.Sender = ZeroAddr\}))$
(3) [SIG] Unchecked CloseRemainderTo	$^{1}\mathbf{Q}(\{currentTxn.CloseRemainderTo\}) \wedge$ $^{2}\mathbf{P}(\{currentTxn.TypeEnum = 1, currentTxn.Type = \text{"pay"}\}) \wedge$ $^{3}\mathbf{I}(currentTxn, Payment) \wedge (^{4}\mathbf{P}(GroupSize > \lvert groupTxns\rvert) \vee$ $\exists txn \in groupTxns, ^{5}\mathbf{P}(\{txn.TypeEnum = 1, txn.Type = \text{"pay"}, txn.Sender = LogicAddr,$ $txn.Index = currentTxn.Index\}) \wedge ^{6}\mathbf{I}(txn, Payment) \wedge ^{7}\mathbf{Q}(\{txn.CloseRemainderTo\}) \wedge$ $^{8}\bigwedge_{t \in groupTxns - \{txn\}} \mathbf{P}(\{t.Sender = RandomAddr \vee t.Sender = ZeroAddr\}))$
(4) [SIG] Unchecked AssetCloseTo	$^{1}\mathbf{Q}(\{currentTxn.AssetCloseTo\}) \wedge$ $^{2}\mathbf{P}(\{currentTxn.TypeEnum = 4, currentTxn.Type = \text{"axfer"}\}) \wedge$ $^{3}\mathbf{I}(currentTxn, AssetTransfer) \wedge (^{4}\mathbf{P}(GroupSize > \lvert groupTxns\rvert) \vee$ $\exists txn \in groupTxns, ^{5}\mathbf{P}(\{txn.TypeEnum = 4, txn.Type = \text{"axfer"},$ $txn.AssetSender = LogicAddr, txn.Sender = ZeroAddr, txn.Index = currentTxn.Index\}) \wedge$ $^{6}\mathbf{I}(txn, AssetTransfer) \wedge ^{7}\mathbf{Q}(\{txn.AssetCloseTo\}) \wedge$ $^{8}\bigwedge_{t \in groupTxns - \{txn\}} \mathbf{P}(\{t.Sender = RandomAddr \vee t.Sender = ZeroAddr\}))$
(5) [APP] Arbitrary update	$^{1}\mathbf{P}(currentTxn.OnCompletion = \text{"UpdateApplication"} \wedge currentTxn.AppID \neq 0) \wedge$ $^{2}\neg\mathbf{R}(\text{"app\_local\_get"}\}) \wedge ^{3}\mathbf{Q}(currentTxn.Sender)$
(6) [APP] Arbitrary delete	$^{1}\mathbf{P}(currentTxn.OnCompletion = \text{"DeleteApplication"} \wedge currentTxn.AppID \neq 0) \wedge$ $^{2}\neg\mathbf{R}(\text{"app\_local\_get"}\}) \wedge ^{3}\mathbf{Q}(currentTxn.Sender)$
(7) [APP] Local state dependency	$^{1}\mathbf{R}(\{\text{"itxn\_submit"}, \text{"app\_global\_put"}, \text{"app\_local\_put"}\}) \wedge$ $\exists account \in local\_accounts, ^{2}\mathbf{P}(\{account \neq currentTxn.Sender, currentTxn.AppID \neq 0,$ $(currentTxn.OnCompletion = \text{"NoOp"} \vee currentTxn.OnCompletion = \text{"CloseOut"})\})$
(8) [APP] Unchecked payment receiver	$^{1}\mathbf{R}(\{\text{"app\_global\_put"}, \text{"app\_local\_put"}\}) \wedge ^{2}\mathbf{P}(\{currentTxn.AppID \neq 0, GroupSize >= 2\}) \wedge$ $\exists txn \in groupTxns, ^{3}\mathbf{P}(\{txn.TypeEnum = 1, txn.Type = \text{"pay"}\}) \wedge$ $^{4}\mathbf{I}(txn, Payment) \wedge ^{5}\mathbf{Q}(\{txn.Receiver\}) \wedge ^{6}\neg\mathbf{Q}(\{txn.Amount\})$
(9) [APP] Unchecked asset receiver	$^{1}\mathbf{R}(\{\text{"app\_global\_put"}, \text{"app\_local\_put"}\}) \wedge ^{2}\mathbf{P}(\{currentTxn.AppID \neq 0, GroupSize >= 2\}) \wedge$ $\exists txn \in groupTxns, ^{3}\mathbf{P}(\{txn.TypeEnum = 4, txn.Type = \text{"axfer"}\}) \wedge$ $^{4}\mathbf{I}(txn, AssetTransfer) \wedge ^{5}\mathbf{Q}(\{txn.AssetReceiver\}) \wedge ^{6}\neg\mathbf{Q}(\{txn.AssetAmount, txn.XferAsset\})$

Table 1: The formal definition of vulnerability detection rules. SIG: smart signature, APP: application.

6 Evaluation

Implementation. `Panda` uses Algorand Indexer [40] to fetch the transaction data and the application data from the blockchain and employs Z3 Theorem Prover v4.11.2 [37] as the constraint solver. We implement `Panda` in Python with more than 7,000 lines of code.

Experimental Setup. Our experiment is performed on a server running Ubuntu 20.04.3 LTS with 128 vCPU AMD EPYC 7H12, 256GB RAM, and three 3.75 TiB Samsung SSD. An Algorand archive node and an Algorand Indexer v2.14.2 are installed on this server. As mentioned in §4, our tool supports several user-defined configuration options. During our experiments, we empirically set the block search depth as 50, the block access count as 3, the Z3 timeout as 30 seconds, and the global timeout as 900 seconds.

Dataset. To construct the dataset, we download the Algorand blockchain data from the first block until block number 25,347,825 (06 Dec 2022 05:59:29 GMT). We process all 969,268,777 transactions in these blocks and finally find 775,848 smart signatures and 543,412 (65,650 unique) applications (including the historical version).

6.1 RQ1: Accuracy of Vulnerability Detection

Although there are some studies on benchmark generation by vulnerability injection, such as LAVA [41], none of them can be directly used to generate benchmarks for the Algorand platform, because it uses a new type of virtual machine, whose opcodes are completely different from those used in other platforms. Moreover, the vulnerabilities defined and detected in this paper are quite different from traditional vulnerabilities. As a result, we develop a new technique to generate the benchmark for the Algorand platform.

Instead of injecting vulnerable code into safe code, we adopt a program synthesis based approach to construct the vulnerable Algorand smart contracts. More precisely, each smart contract consists of multiple execution paths made up of three types of basic blocks. The first is security check basic blocks, which perform checks on specific security semantics. These basic blocks check some critical transaction parameters such as `CloseRemainderTo`. To generate security check basic blocks, we developed an automated tool that accepts the vulnerability type as a parameter to generate the corresponding check code. The other two types of basic blocks are used to check the semantic correctness of the symbolic execution tool. One is generated by the automated program developed by ourselves based on Algorand semantics. The other includes hand-constructed basic blocks which are used to cover more corner tests. Finally, we connect these basic blocks together and output them as test files by recursively constructing data streams.

We use the aforementioned method to construct a benchmark dataset consisting of 3,500 safe cases and 7,500 vulnerable cases. The evaluation results show that all the vulnerable and safe samples can be correctly classified by `Panda`.

6.2 RQ2: Prevalence of Vulnerabilities in Smart Signatures

Among all 775,848 smart signatures, 512,631 contain `Validator` and 263,217 do not. For smart signatures that contain `Validator`, we will use the technique mentioned in §4.3 to merge the `Validator` and the smart signature into a new smart signature before performing symbolic execution.

6.2.1 Overall Results

	Whether contain Validator	
	YES	NO
Type	Vulnerable (%*)	Vulnerable (%*)
Unchecked transaction fees	15,539 (3.03%)	23,251 (8.83%)
Unchecked rekey_to	751 (0.15%)	8,713 (3.31%)
Unchecked close_remainder_to	42,084 (8.21%)	4,509 (1.71%)
Unchecked asset_close_to	900 (0.18%)	3,206 (1.22%)
Total	57,120 (11.14%)	23,395 (8.89%)

Table 2: Results of smart signatures

Table 2 shows the overall results. It is worth noting that a total of 57,120 (11.14%) smart signatures containing `Validator` are flagged as vulnerable, while the number of smart signatures without `Validator` is 23,395 (8.89%). The total number of vulnerable cases is 80,515 (10.38%). From the experimental results, we know that these four vulnerabilities are prevalent in smart signatures, revealing the urgency of identifying and preventing such vulnerabilities. Interested readers may refer to §B.2 and §B.3 for the case study.

6.2.2 Manual Verification

To verify the results, we randomly selected 500 samples marked as vulnerable and 100 marked as safe based on the vulnerability types and the number of opcodes. Specifically, we sort all smart signatures according to the number of opcodes and divide them into three groups, and randomly select samples from each interval. Since it is time-consuming to traverse all execution paths to verify the secure examples, for safe cases with more than 500 opcodes, we only randomly select 10 execution paths for manual verification. The results show that there is only one false negative case, and all other samples are correctly classified. This false negative case calls `Validator` with implicit code formatting (see §7). We also verified the 10 vulnerable smart signatures with the highest balance, and all of these are true positives.

6.3 RQ3: Prevalence of Vulnerabilities in Applications

As mentioned in §4, `Panda` supports two runtime modes for applications. We analyze all the 543,412 (65,650 unique) applications (including the historical version) using the symbolic mode and then collect all the 99,142 (14,184 unique) applications that have not been deleted and apply the constant mode to analyze them. The reason of using use the symbolic mode to analyze deleted applications is that recovering the global state of a deleted application is currently difficult (Algorand Indexer currently does not support such functionality). In the following, we will refer to the 543,412 applications as off-chain applications and the 99,142 applications that have not been deleted as on-chain applications.

6.3.1 Overall Results

Table 3 and 4 report the overall results. We can see that vulnerabilities in Algorand applications are widespread. Notably, 4,008 (4.04%) on-chain applications and 150,676 (27.73%) off-chain applications are marked as vulnerable. The arbitrary update and arbitrary delete vulnerabilities are the most prevalent, and we provide a case study in §B.1.

	off-chain	
Type	Vulnerable (%*)	Unique (%*)
Arbitrary update	91,246 (16.79%)	454 (0.69%)
Arbitrary delete	97,908 (18.02%)	437 (0.67%)
Force clear state	10,749 (1.98%)	441 (0.67%)
Unchecked payment receiver	1,570 (0.29%)	98 (0.15%)
Unchecked asset receiver	43,066 (7.93%)	141 (0.21%)
Total	150,676 (27.73%)	987 (1.50%)

Table 3: Results for off-chain applications

	on-chain	
Type	Vulnerable (%*)	Unique (%*)
Arbitrary update	1,420 (1.43%)	147 (1.04%)
Arbitrary delete	2,590 (2.61%)	167 (1.18%)
Force clear state	1,360 (1.37%)	141 (0.99%)
Unchecked payment receiver	710 (0.72%)	48 (0.34%)
Unchecked asset receiver	123 (0.12%)	60 (0.42%)
Total	4,008 (4.04%)	364 (2.57%)

Table 4: Results for on-chain applications

6.3.2 Manual Verification

To verify the results, we randomly selected 100 samples marked as vulnerable and 100 marked as safe based on the vulnerability types and the number of opcodes. As before, we divide the number of opcodes into three groups and randomly select samples from each interval. For safe samples, we only check cases with less than 500 opcodes. We also verified 10 vulnerable applications with the highest number of duplicates. The results show that all samples are correctly classified.

6.4 RQ4: The Presence of Attacks

Index	Vulnerability Type	Mode	Suspicious Attacks
1	Unchecked transaction fee	Signature	3
2	Unchecked RekeyTo	Signature	5
3	Unchecked CloseRemainderTo	Signature	44,428
4	Unchecked AssetCloseTo	Signature	1,284
5	Arbitrary update	Application	11
6	Arbitrary delete	Application	7,194

Table 5: Suspicious Attacks

We developed 6 heuristic rules to check all the transactions on the blockchain to find possible attacks exploiting the vulnerabilities. The results are shown in table 5. Specifically, for the vulnerabilities of smart signatures with index 1-4, we first find the transaction with the corresponding parameters (or transaction fee greater than 100,000 `microAlgos` for index 1) and then determine whether there is a corresponding vulnerability in the smart signature attached to the transaction. For example, suppose a transaction is attached to a smart signature with an unchecked `RekeyTo` vulnerability and the `RekeyTo` of this transaction is set. In this case, we regard this transaction as a suspicious attack. For the two vulnerabilities of applications with index 5-6, we first find the application `delete` or `update` transaction and then determine whether the targeting application is vulnerable and whether the transaction's sender is not the application's creator. For example, suppose there is an application `update` transaction targeting a vulnerable application and the transaction sender is not the application's creator. In that case, we consider this transaction a suspicious attack. Detecting the other three types of attacks requires replaying transactions at the specific block height and performing analysis during execution. We leave building a replaying system to detect such attacks in future work.

6.5 RQ5: The Performance of `Panda`

Figure 4 reports the analysis time by running `Panda`. Note that the number of applications shown in this figure is the result before deduplication. The median and average analysis times for applications are 15 seconds and 67 seconds, while the results for smart signatures are 19 seconds and 35 seconds, respectively. Note that the total analysis time of applications and smart signatures exceeds 500 days. Due to the heavy analysis workload, we developed a script to let `Panda` run in parallel to process these smart contracts.

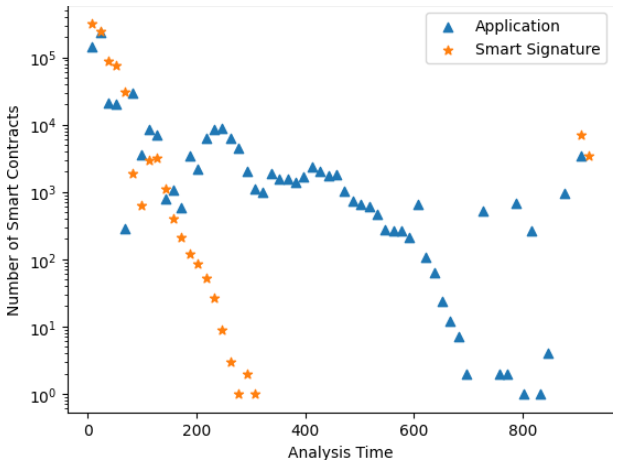

Figure 4: The analysis time of applications and smart signatures. A point is taken at an interval of 15 seconds.

7 Discussion

Due to the functional limitations of the Z3 symbolic execution engine, some opcodes (such as the b+ opcode) in `Panda` cannot handle symbolic values. The symbolic executor will return immediately when operands of these opcodes are symbolic values, and other execution paths that do not contain these opcodes or the operands of these opcodes are constant values can be executed successfully. There are 8,391 out of 543,412 applications and 1 out of 775,848 smart signatures containing such opcodes.

Another limitation of our tool is in the process of identifying the `Validator` in the smart signature. To be specific, our tool uses pattern matching to identify the `AppID` corresponding to the `Validator`. There are a few smart signatures that use an implicit code format to invoke the `Validator`; that is, the `AppID` of the `Validator` is dynamically generated rather than hardcoded in the smart signature. As a consequence, there are 25,721(3.3%) cases where our method cannot identify the `Validator`.

8 Related Work

Algorand Analysis Besides some white papers on the underlying protocol and architecture of Algorand [5, 42–45], there are some studies on smart contracts or the security of the Algorand platform. For example, Bartoletti et al. [46] developed a formal model of Algorand smart signatures to prove some basic properties of the Algorand blockchain. Alturki et al. [47] presented a model of the Algorand consensus protocol and outlined the specification and formal proof of its asynchronous safety.

Security Analysis of Ethereum Smart Contracts The security of Ethereum smart contracts has received great attention [48–53], and researchers have developed many vulnerability detection tools for them. For example, Luu et al. [35] developed the first symbolic execution-based tool to detect vulnerabilities in Ethereum smart contracts. Sunbeom et at. [54] combine symbolic execution with a language model for vulnerable transaction sequences to prioritize program paths that are likely to reveal vulnerabilities. In addition to symbolic execution [55], fuzzing [56, 57], pattern recognition [58–63], formal analysis [64–68] and machine learning [69] are also used to identify vulnerabilities. Brent et al. [58] present a security analysis framework named Vandal which consists of an analysis pipeline that converts low-level EVM bytecode to semantic logic relations. Users of this framework can express security analyses in a declarative fashion. The work presented in [65] reviews several static analysis methodologies and discusses EtherTrust, the first proof of concept for reachability analysis based on Horn clauses. Schneidewind et al. present the first sound and automated static analyzer for EVM bytecode named eThor [66], which is built on an abstraction of the small-step EVM bytecode semantics [67].

Comparison with Ethereum analysis tools Since AVM and EVM are totally different, we had to re-architect a new symbolic execution engine for AVM. Specifically, `Panda` is significantly different from previous symbolic execution tools for Ethereum (e.g. OYENTE) in the following 4 aspects. First, since Algorand supports two different data types, `Panda` is equipped with operand type checking and type inference systems. In contrast, Ethereum only has one data type (i.e. Uint), and thus the analysis tools for it do not involve the processing of data types. Second, due to the diversity of data types and the complexity of functionalities, the opcodes supported by AVM are far more complex than those of EVM. To this end, we designed new handlers for these opcodes. Third, compared to Ethereum smart contracts, Algorand also supports smart signatures. To deal with the situation where the `Validator` is included in the smart signature, we design the smart signature merging technique for `Panda`. Lastly, the vulnerabilities in Ethereum smart contracts and those in Algorand smart contracts are totally different. We identify 9 types of vulnerabilities in Algorand smart contracts after conducting an in-depth analysis of the Algorand platform. Most importantly, compared to the vulnerabilities in Ethereum smart contracts, vulnerability detection in Algorand smart contracts involves more complex program semantics. To this end, we designed a set of sound detection rules for these vulnerabilities.

9 Disclosure Process

We sent all the evaluation results to the Algorand foundation and work with them to help developers fix these vulnerabilities. Due to the large number of vulnerable contracts, we only deal with applications that have not been deleted and prioritize smart contracts that have large deposits.

A number of developers have confirmed the reported vulnerabilities. For example, the NFT auction platform named ALGOxNFT [14] has many vulnerable escrow accounts. Every auction item on this platform is implemented as a signature account. However, there is a vulnerability in the smart signature that allows attackers to steal the NFT and all auction deposits in the account. `Panda` reported 41,204 such vulnerable signature accounts with a total trade volume of 2,158,009 `Algos` (worth more than 1 million dollars according to the average price of `Algo`). We reached out to the developers directly and helped them fix the bugs, and received a 10,000 `Algos` bug bounty. Another impactful example is a vulnerable liquidity pool of FXDX [13] with a deposit of 541,456 `Algos` (worth about $120,000 at that time). `Panda` successfully detected this vulnerability, and we reported this case to the Algorand DevRel team. They quickly notified the developers of FXDX, and the vulnerability was fixed. Interested readers may refer to B.3 for more technical details.

We plan to deliver some lectures on Algorand smart contract security and work with the Algorand team to integrate `Panda` into the AlgoSDK to help more developers improve the security of their smart contracts.

10 Conclusion

We identify 9 types of vulnerabilities on the Algorand platform and formally defined the detection rules. We design and develop `Panda`, a symbolic execution-based framework to accurately detect these vulnerabilities. The large-scale evaluation based on all smart contracts on the blockchain shows the prevalence of these vulnerabilities in the Algorand ecosystem.

Acknowledgement

We would like to thank all anonymous reviewers for their helpful suggestions to improve the paper. This work was partially supported by Hong Kong RGC Projects (No. PolyU15219319, and No. PolyU15224121).

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," Decentralized Business Review, p. 21260, 2008.

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

[3] J. Sedlmeir, H. U. Buhl, G. Fridgen, and R. Keller, "The energy consumption of blockchain technology: beyond myth," Business & Information Systems Engineering, vol. 62, no. 6, pp. 599–608, 2020.

[4] Y. Hao, Y. Li, X. Dong, L. Fang, and P. Chen, "Performance analysis of consensus algorithm in private blockchain," in 2018 IEEE Intelligent Vehicles Symposium (IV). IEEE, 2018, pp. 280–285.

[5] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling byzantine agreements for cryptocurrencies," in Proceedings of the 26th symposium on operating systems principles, 2017, pp. 51–68.

[6] "Algorand grants program." [Online]. Available: https://algorand.foundation/grants-program/grant-awardees

[7] Wayne Jones, "Algorand is growing in transaction rate as more use cases get introduced," Mar. 2022. [Online]. Available: https://crypto.news/algorand-transaction-rate-use-cases-get/

[8] "Algorand adds over 6 million new accounts in 2022," 2022. [Online]. Available: https://finbold.com/algorand-adds-over-6-million-new-accounts-in-2022/

[9] Phil Daian, "Analysis of the dao exploit," 2016. [Online]. Available: https://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/

[10] "Algorand atomic transfers." [Online]. Available: https://developer.algorand.org/docs/get-details/atomic_transfers/

[11] "Algorand standard assets." [Online]. Available: https://developer.algorand.org/docs/get-details/asa/

[12] Tinyman, "Full Technical Report on Attacks," Jan. 2022. [Online]. Available: https://tinymanorg.medium.com/full-technical-report-on-attacks-18e3c5e89c5f

[13] "Fxdx exchange." [Online]. Available: https://fxdx.exchange/

[14] "Algoxnft." [Online]. Available: https://algoxnft.com/

[15] "Algorand account overview." [Online]. Available: https://developer.algorand.org/docs/get-details/accounts/

[16] "Ethereum accounts." [Online]. Available: https://ethereum.org/en/whitepaper/#ethereum-accounts

[17] "Algorand contract account." [Online]. Available: https://developer.algorand.org/docs/get-details/dapps/smart-contracts/smartsigs/modes/#contract-account

[18] "Algorand virtual machine." [Online]. Available: https://developer.algorand.org/docs/get-details/dapps/avm/

[19] "The smart contract language." [Online]. Available: https://developer.algorand.org/docs/get-details/dapps/avm/teal/

[20] "Algorand opcodes." [Online]. Available: https://developer.algorand.org/docs/get-details/dapps/avm/teal/opcodes/

[21] "Algorand smart contracts." [Online]. Available: https://developer.algorand.org/docs/get-details/dapps/smart-contracts/#smart-contracts

[22] "Algorand smart signatures." [Online]. Available: https://developer.algorand.org/docs/get-details/dapps/smart-contracts/#smart-signatures

[23] "PyTeal: Algorand Smart Contracts in Python." [Online]. Available: https://pyteal.readthedocs.io/en/stable/

[24] "The lifecycle of a smart contract." [Online]. Available: https://developer.algorand.org/docs/get-details/dapps/smart-contracts/apps/#the-lifecycle-of-a-smart-contract

[25] "Algorand oncomplete constants." [Online]. Available: https://developer.algorand.org/docs/get-details/dapps/avm/teal/specification/#oncomplete

[26] "Delegate signature authority." [Online]. Available: https://developer.algorand.org/docs/get-details/dapps/smart-contracts/smartsigs/modes/#delegated-approval

[27] "Algorand transaction types." [Online]. Available: https://developer.algorand.org/docs/get-details/transactions/

[28] "Algorand payment transaction." [Online]. Available: https://developer.algorand.org/docs/get-details/transactions/transactions/#payment-transaction

[29] "Algorand rekeying." [Online]. Available: https://developer.algorand.org/docs/get-details/accounts/rekey/

[30] "Algorand asset transfer transaction." [Online]. Available: https://developer.algorand.org/docs/get-details/transactions/transactions/#asset-transfer-transaction

[31] M. Kaleem, A. Mavridou, and A. Laszka, "Vyper: A security comparison with solidity based on common vulnerabilities," in 2020 2nd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS). IEEE, 2020, pp. 107–111.

[32] "Algorand transaction fee." [Online]. Available: https://developer.algorand.org/docs/get-details/transactions/#fees

[33] "Algorand python sdk." [Online]. Available: https://github.com/algorand/py-algorand-sdk/

[34] "Algorand parameter tables." [Online]. Available: https://developer.algorand.org/docs/get-details/parameter_tables/#smart-signature-constraints

[35] 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, 2016, pp. 254–269.

[36] C. Cadar, D. Dunbar, D. R. Engler et al., "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs." in OSDI, vol. 8, 2008, pp. 209–224.

[37] Microsoft Corporation, "The z3 theorem prover." [Online]. Available: https://github.com/Z3Prover/z3

[38] "Lazy binding." [Online]. Available: http://www.qnx.com/developers/docs/7.0.0/index.html#com.qnx.doc.neutrino.prog/topic/devel_Lazy_binding.html

[39] "Algorand transaction reference." [Online]. Available: https://developer.algorand.org/docs/get-details/transactions/transactions/

[40] "Algorand indexer." [Online]. Available: https://github.com/algorand/indexer

[41] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "Lava: Large-scale automated vulnerability addition," in 2016 IEEE Symposium on Security and Privacy (SP). IEEE, 2016, pp. 110–121.

[42] M. Drijvers, S. Gorbunov, G. Neven, and H. Wee, "Pixel: Multi-signatures for consensus." in USENIX Security Symposium, 2020, pp. 2093–2110.

[43] D. Leung, A. Suhl, Y. Gilad, and N. Zeldovich, "Vault: Fast bootstrapping for the algorand cryptocurrency," Cryptology ePrint Archive, 2018.

[44] J. Chen, S. Gorbunov, S. Micali, and G. Vlachos, "Algorand agreement: Super fast and partition resilient byzantine agreement," Cryptology ePrint Archive, 2018.

[45] J. Chen and S. Micali, "Algorand," arXiv preprint arXiv:1607.01341, 2016.

[46] M. Bartoletti, A. Bracciali, C. Lepore, A. Scalas, and R. Zunino, "A formal model of algorand smart contracts," in International Conference on Financial Cryptography and Data Security. Springer, 2021, pp. 93–114.

[47] M. A. Alturki, J. Chen, V. Luchangco, B. Moore, K. Palmskog, L. Peña, and G. Roşu, "Towards a verified model of the algorand consensus protocol in coq," in International Symposium on Formal Methods. Springer, 2019, pp. 362–367.

[48] T. Chen, Z. Li, K. Fang, X. Luo, T. Wang, Y. Zhang, H. Zhu, X. Wang, H. Li, and X. Zhang, "Sigrec: Automatic recovery of function signatures in smart contracts," IEEE Transactions on Software Engineering, vol. 48, no. 8, 2022.

[49] K. Li, J. Chen, X. Liu, Y. Tang, X. Wang, and X. Luo, "As strong as its weakest link: How to break blockchain dapps at rpc service," in Proc. NDSS, 2021.

[50] T. Chen, R. Cao, T. Li, X. Luo, G. Gu, Y. Zhang, Z. Liao, H. Zhu, G. Chen, Z. He, Y. Tang, X. Lin, and X. Zhangg, "Soda: A generic online detection framework for smart contracts," in Proc. NDSS, 2020.

[51] T. Chen, Y. Zhang, Z. Li, X. Luo, T. Wang, R. Cao, X. Xiao, and X. Zhang, "Tokenscope: Automatically detecting inconsistent behaviors of cryptocurrency tokens in ethereum," in Proc. CCS, 2019.

[52] T. Chen, Z. Liao, H. Zhu, X. Luo, Z. He, J. Chen, T. Zhang, and X. Zhang, "Large-scale empirical study of inline assembly on 7.6 million ethereum smart contracts," IEEE Transactions on Software Engineering, vol. 49, no. 2, 2023.

[53] T. Chen, Y. Zhu, Z. Li, J. Chen, X. Li, X. Luo, X. Lin, and X. Zhang, "Understanding ethereum via graph analysis," in Proc. INFOCOM, 2018.

[54] S. So, S. Hong, and H. Oh, "{SmarTest}: Effectively hunting vulnerable transaction sequences in smart contracts through language {Model-Guided} symbolic execution," in 30th USENIX Security Symposium (USENIX Security 21), 2021, pp. 1361–1378.

[55] P. Zheng, Z. Zheng, and X. Luo, "Park: Accelerating smart contract vulnerability detection via parallel-fork symbolic execution," in Proc. ISSTA, 2022.

[56] B. Jiang, Y. Liu, and W. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2018, pp. 259–269.

[57] J. Su, H. Dai, L. Zhao, Z. Zheng, and X. Luo, "Effectively generating vulnerable transaction sequences in smart contracts with reinforcement learning-guided fuzzing," in Proc. ASE, 2022.

[58] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, "Vandal: A scalable security analysis framework for smart contracts," arXiv preprint arXiv:1809.03981, 2018.

[59] 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.

[60] 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.

[61] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, "Reguard: finding reentrancy bugs in smart contracts," in 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion). IEEE, 2018, pp. 65–68.

[62] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen, "Defining smart contract defects on ethereum," IEEE Transactions on Software Engineering, vol. 48, no. 1, pp. 327–345, 2020.

[63] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen, "Defectchecker: Automated smart contract defect detection by analyzing evm bytecode," IEEE Transactions on Software Engineering, vol. 48, no. 7, pp. 2189–2207, 2021.

[64] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu et al., "Kevm: A complete formal semantics of the ethereum virtual machine," in 2018 IEEE 31st Computer Security Foundations Symposium (CSF). IEEE, 2018, pp. 204–217.

[65] I. Grishchenko, M. Maffei, and C. Schneidewind, "Foundations and tools for the static analysis of ethereum smart contracts," in International Conference on Computer Aided Verification. Springer, 2018, pp. 51–78.

[66] C. Schneidewind, I. Grishchenko, M. Scherer, and M. Maffei, "ethor: Practical and provably sound static analysis of ethereum smart contracts," in Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, 2020, pp. 621–640.

[67] I. Grishchenko, M. Maffei, and C. Schneidewind, "A semantic framework for the security analysis of ethereum smart contracts," in International Conference on Principles of Security and Trust. Springer, 2018, pp. 243–269.

[68] C. Schneidewind, M. Scherer, and M. Maffei, "The good, the bad and the ugly: Pitfalls and best practices in automated sound static analysis of ethereum smart contracts," in International Symposium on Leveraging Applications of Formal Methods. Springer, 2020, pp. 212–231.

[69] J. He, M. Balunović, N. Ambroladze, P. Tsankov, and M. Vechev, "Learning to fuzz from symbolic execution with application to smart contracts," in Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, 2019, pp. 531–548.

A Detection Example

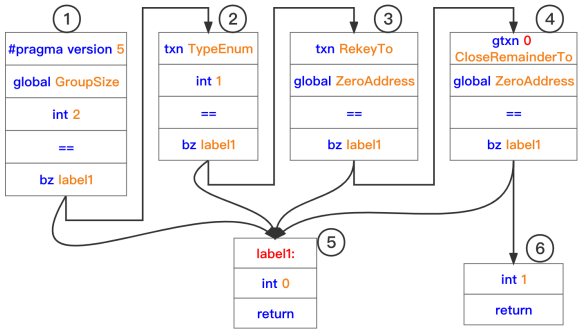

Figure 5: The control flow graph of the detection example

To facilitate understanding, we give a sample program to show how the detection rules are applied. Figure 5 shows the control flow graph of the sample program. This program is a simple smart signature, and we have numbered each basic block in the figure. During the running process of `Panda`, the control flow graph is traversed to find feasible paths. Specifically, only paths with solvable path constraints and non-zero

Basic Block Number	Semantic Action
1	version = 5 path_constraints.add(global.GroupSize == 2) path_constraint_variable.add(global.GroupSize)
2	path_constraints.add(current_transaction.TypeEnum == 1) path_constraint_variable.add(current_transaction.TypeEnum)
3	path_constraints.add(current_transaction.RekeyTo == ZeroAddress) path_constraint_variable.add(current_transaction.RekeyTo)
4	path_constraints.add(gtxn[0].CloseRemainderTo == ZeroAddress) path_constraint_variable.add(gtxn[0].CloseRemainderTo) groupTxns.add(0)
5	The program reaches a feasible path and the vulnerability detection function will be executed.
6	The program reaches an infeasible path.

Table 6: Semantics of the example program

Index \ Parts	1	2	3	4	5	6	7	8	Results
1	✓	✓	✓	✓	✓	-	-	-	True
2	✗	✓	✓	✓	✓	✓	-	-	False
3	✓	✓	✓	✓	✓	✓	✗	✓	True
4	✓	✗	✓	✓	✗	✓	✓	✓	False

Table 7: The detection results. ✗ means that the vulnerability detection result for this part is false while ✓ means it is true.

elements at the top of the stack when the program executes the `return` instruction are feasible, in this case, there is only one feasible path, that is 1->2->3->4->6.

In table 6, we list the semantics corresponding to the instructions within each basic block. Specifically, basic blocks 1-5 contain codes for checking transaction parameters. These codes will be added to `path_constraints`, and the corresponding symbolic variables will be added to `path_constraint_variables`. In addition, the group transaction with index 0 is accessed using the `gtxn` bytecode in basic block 5, so the index 0 is added to `groupTxns`.

Table 7 shows the vulnerability detection results of this execution trace. First, there is no check on transaction fees in the example program, so there is an unchecked transaction fee vulnerability. Second, the `RekeyTo` parameter of the current transaction is checked in basic block 3 of the sample program, so part 1 of rule 2 is false, and therefore there is no unchecked `RekeyTo` vulnerability. Next, there is no check on the `CloseRemainderTo` parameter of the current transaction in the sample program, and the group size (assigned to 2 in the basic block 1) is larger than the size of `groupTxns` (only index 0 is accessed in basic block 4, so the size is 1). This means that there is at least one transaction in the atomic transaction group whose `CloseRemainderTo` parameter has not been checked by any code, and in this example, the index of this transaction is 1. As a result, there is an unchecked `CloseRemainderTo` vulnerability. Finally, the sample program defines the type of the current transaction as payment in basic block 2 (i.e., `TypeEnum` equals to 1), so part 2 of rule 4 is false, so there is no unchecked `AssetCloseTo` vulnerability.

B Case Study

During our manual analysis of the experimental results, we found many vulnerable smart contracts. In the following, we will analyze several typical examples in detail.

B.1 Unexpected Delete and Update Operation

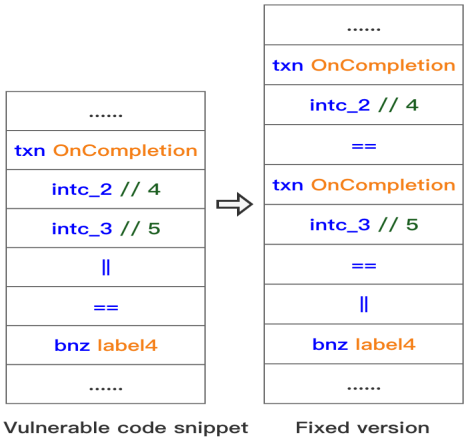

Figure 6: case 1

In figure 6, we give an example code of a smart contract that includes both arbitrary update and arbitrary deletion vulnerabilities. The reason for the vulnerability in this code is apparently due to a programming mistake by the developer. Specifically, the code has no correct constraint on the `OnCompletion` parameter. Note that we give the corresponding fixed code (on the right of the arrow) to help understand the cause of the vulnerability. Note that this vulnerable example has a duplicate of 333 on the blockchain.

B.2 Unchecked group size

In figure 7, we give a simplified example code for a smart signature that includes vulnerabilities for unchecked transaction fees and unchecked transaction parameters (corresponding to indexes 1-4 in table 1). This code is vulnerable because the group size is not a fixed value, and at least one of the transactions whose parameters have not been checked. As before, we give a fixed version of this code. Remember that we have to specify the group size explicitly and check all the parameters of each of these transactions in smart signatures or in the `Validator`.

B.3 Validator can be bypassed

During the evaluation process, we found that a large number of smart signatures used an incorrect way to call the

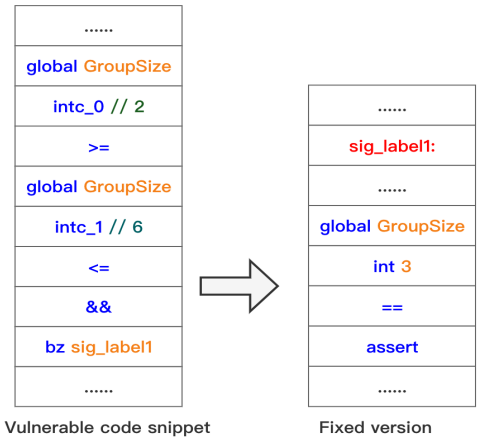

Figure 7: case 2

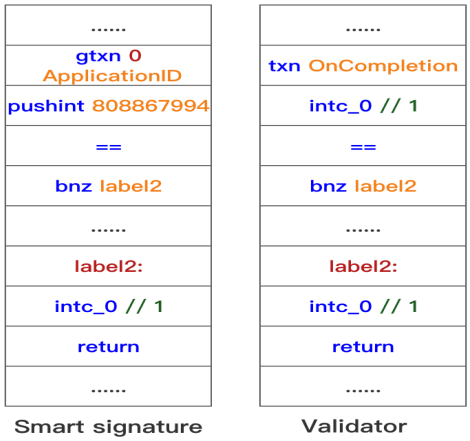

Figure 8: case 3

`Validator`. As shown in figure 8, the logic of these smart signatures restricts the `Validator`'s application ID but does not restrict the `Oncomplete` parameter. Therefore, an attacker can use `opt-in`, `close-out`, or `clear-state` transactions to bypass the validation logic. This is because the main logic of the `Validator` is located in the `NoOp` branch, while other branches do not contain validation code. `Panda` has reported a large number of smart signatures with this vulnerability pattern such as the 41,204 vulnerable escrow accounts of ALGOxNFT [14] and the vulnerable liquidity pool of FXDX [13] mentioned in §9.