AUTOMATED VERIFICATION TECHNIQUES FOR SOLANA SMART CONTRACTS

An Undergraduate Research Scholars Thesis

by

TIEN N. TAVU

Submitted to the LAUNCH: Undergraduate Research office at
Texas A&M University
in partial fulfillment of requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by
Faculty Research Advisor: Dr. Jeff Huang

May 2022

Major: Computer Science & Engineering

RESEARCH COMPLIANCE CERTIFICATION

Research activities involving the use of human subjects, vertebrate animals, and/or biohazards must be reviewed and approved by the appropriate Texas A&M University regulatory research committee (i.e., IRB, IACUC, IBC) before the activity can commence. This requirement applies to activities conducted at Texas A&M and to activities conducted at non-Texas A&M facilities or institutions. In both cases, students are responsible for working with the relevant Texas A&M research compliance program to ensure and document that all Texas A&M compliance obligations are met before the study begins.

I, Tien N. Tavu, certify that all research compliance requirements related to this Undergraduate Research Scholars thesis have been addressed with my Research Faculty Advisor prior to the collection of any data used in this final thesis submission.

This project did not require approval from the Texas A&M University Research Compliance & Biosafety office.

TABLE OF CONTENTS

ABSTRACT

Automated Verification Techniques for Solana Smart Contracts

Tien N. Tavu
Department of Computer Science & Engineering
Texas A&M University

Research Faculty Advisor: Dr. Jeff Huang
Department of Computer Science & Engineering
Texas A&M University

Solana has been a relatively new blockchain platform that has gained popularity due to its quick transaction times and low transaction fees. However, the focus is mainly seen in their "smart contracts" – an automatically-enforced agreement under an on-chain program between an individual with financial implications involved. Due to the nature of the platform being relatively new, there has been no foundation related to the security concerns of developing these programs, but such programs have been continually deployed daily without any security considerations.

During the investigation of real-world smart contracts, we found that there were several common vulnerabilities – missing ownership checks, missing signer checks, the signed invocation of programs, and the underflow and overflow of arithmetic operations. The mentioned vulnerabilities became the baseline for us to develop verification techniques in identifying them in real-world smart contracts. Furthermore, it became a goal to develop a static analysis tool in Rust that combines all the algorithms into a single static analysis tool, leveraging the MIR functionality provided by Rust. The results conveyed that the tool was able to reliably find sensitive instructions it deemed to be insecure. Even though there were several insignificant

results, the initial verification techniques are valid in this early stage of development. Developers who wish to develop Solana smart contracts should use these verification techniques in practice before on-chain deployment as an initial benchmark for security concerns.

DEDICATION

I would like to dedicate my family members to allow me to pursue my passion for Computer Science, in addition to supporting me in pursuing this passion.

Additionally, I would like to thank Dr. Jeff Huang for allowing me an opportunity to participate in undergraduate research and showcasing my abilities to contribute to a field of interest.

Moreover, I would like to acknowledge the Origin-Oriented (O2) Programming Lab, who were a part of this research and supported me throughout the process.

Lastly, I would like to mention all the friends, peers, and instructors who were able to contribute to who I am today.

ACKNOWLEDGEMENTS

Contributors

I would like to thank my faculty advisor, Dr. Jeff Huang, and the research lab members in Gang Zhao, Siwei Cui, and Yifei Gao, for their guidance and support throughout the course of this research. Thanks also go to my friends and colleagues and the department faculty and staff for making my time at Texas A&M University a great experience.

The static analysis tool and verification techniques were a contribution from all the members of the O2 research lab. The source code from the smart contracts used was open-sourced and available on GitHub with an MIT License. All other work conducted for the thesis was completed by the student independently.

Funding Sources

This work was also made possible in part by the National Science Foundation under Grant Number 1552935. Its contents are solely the responsibility of the authors and do not necessarily represent the official views of the National Science Foundation.

1. INTRODUCTION

Research in Computer Science has recently focused on the field of blockchain – a technology focused as a decentralized, distributed, and public digital ledger of transactions recorded and "chained" as a sequence of blocks. Bitcoin is one of the well-known blockchain platforms, but new technologies such as Ethereum and Solana have emerged as competitors. Ethereum and Solana leverage the blockchain to store code of automated programs based on an agreement between a buyer known as "smart contracts," in addition to the original intentions of supporting a decentralized payment network as seen in Bitcoin [1, 2].

The term smart contract was coined by Nick Szabo, who outlined his ideas as analogous to a vending machine: with the right set of inputs, a desired output is reached [6]. The logic of a vending machine is programmed to accept an input of a certain amount of funds, and in addition to a selection for a specific product, will dispense any change and the said product as an output. A smart contract follows the exact logic of an agreement between a buyer that is automatically enforced programmatically, removing the need for an intermediary such as financial institutions. If the necessary fees are paid with the transaction from a smart contract, the transaction would be executed and stored in the blockchain network.

Ethereum, known as the second-most popular platform behind Bitcoin by market capitalization, popularized the technology and term of smart contracts to blockchain platforms. Developers have deployed many applications built with Solidity – the programming language used for Ethereum – in the blockchain for the past few years since its inception. However, developers and end-users have raised concerns involving the Ethereum network, as one of the notorious disadvantages of Ethereum is the transaction times and fees. The network announced plans for solving the issues in a roadmap called “The Merge,” such as the transition to a Proof-of-Stake network, but the overall implications have yet to be seen.

Another blockchain platform that has smart contract capabilities is Solana, whose popularity has tremendously grown in terms of market cap and usage. One of the few driving reasons for its growth is its nature as a competitor to the Ethereum network in achieving higher transaction speeds at a lower cost, mitigating the main concerns that Ethereum is currently experiencing from its Proof-of-Stake model. Additionally, the platform uses Rust as its primary programming language for smart contracts, which is known as a multipurpose language designed for performance and safety, in comparison to Ethereum’s Solidity programming language which is recognized for only smart contracts.

The subfield of smart contracts in blockchain networks is still relatively new, with vast potential for impact and growth. Examples of smart contracts can include services provided by centralized financial institutions such as banks – borrowing, lending, and trading, to name a few. Furthermore, Non-Fungible Tokens (NFTs) have been a controversial topic but have utility for areas in real estate and ticketing. However, smart contracts are only a breakthrough – being massively adopted and deployed in the field of blockchain without any repercussions, smart contracts and their developers have faced adversities in handling scaling and security issues.

Solana is a relatively new blockchain platform with a minimal foundation on scalability and security issues, which is particularly vulnerable for developers in deploying smart contracts to the network. With the prominence of the blockchain field and the Solana ecosystem in general, we explored several different vulnerabilities and ways to identify such vulnerabilities. This paper will discuss several vulnerability patterns found from previously deployed smart contracts and a set of automated verification techniques to identify such patterns. Additionally, we question its effectiveness with the results that were obtained by combining the verification techniques within a single static analysis tool developed in Rust on real-world smart contracts. With these techniques, we hope to improve the security of smart contracts in Solana which would lead to the mass adoption and usage of the applications in the future.

2. OVERVIEW OF VULNERABILITIES

The list of vulnerabilities found in smart contracts is growing with an ongoing process to discover such exploits. A few vulnerabilities can be generally found in all smart contract platforms such as Ethereum's Solidity programming language, while a few other vulnerabilities are unique to the Solana ecosystem.

The vulnerabilities that are unique to the Solana ecosystem require a general understanding of how Solana smart contract development works. The platform utilizes "accounts" as its main primitive type to store records in its ledger for data or executable programs such as a smart contract. The notable fields inside this type can be seen in Figure 2.1.

```
pub struct AccountInfo<'a> {
    /// Public key of the account
    pub key: &'a Pubkey,
    /// Was the transaction signed by this account's public key?
    pub is_signer: bool,
    /// The lamports in the account.  Modifiable by programs.
    pub lamports: Rc<RefCell<&'a mut u64>>,
    /// The data held in this account.  Modifiable by programs.
    pub data: Rc<RefCell<&'a mut [u8]>>,
    /// Program that owns this account
    pub owner: &'a Pubkey,
    /// This account's data contains a loaded program (and is now read-only)
    pub executable: bool,
    . . .
}
```

Figure 2.1. Code snippet extracted from the Solana source code related to accounts.

The fields that were highlighted in the code snippet are crucial to understanding the specific vulnerabilities that are found in the Solana ecosystem.

2.1 Missing Ownership Checks

The most vital vulnerabilities are smart contracts that do not include ownership checks. Each program that is deployed to a Solana cluster would be available to clients through a program ID – an address stored as a `Pubkey` type that is used to reference the program for transactions [5]. An account has an `owner` field that corresponds to a single program ID that can write to the specific account, as the program "owns" the rights to the account in storing data [3]. Without any checks to determine whether a supplied account is owned by the program, the program is essentially dealing with untrusted data that can be spoofed by a user.

Take a `withdraw` function as an example, which is a sensitive instruction that any user can execute to withdraw funds from a savings account stored in a personal wallet. The function shown in Figure 2.2 is problematic, as the function never implements checks relating to the owner of the accounts being passed in.

```rust
fn withdraw(program_id: &Pubkey, accounts: &[AccountInfo], amount: u64)
  -> ProgramResult {
    let account_info_iter = &mut accounts.iter();
    let wallet_info = next_account_info(account_info_iter)?;
    let authority_info = next_account_info(account_info_iter)?;
    let destination_info = next_account_info(account_info_iter)?;
    let wallet = Wallet::deserialize(&mut &(*wallet_info.data).borrow_mut()[..])?;

    if amount > **wallet_info.lamports.borrow_mut() {
        return Err(ProgramError::InsufficientFunds);
    }

    **wallet_info.lamports.borrow_mut() -= amount;
    **destination_info.lamports.borrow_mut() += amount;
    . . .
}
```

Figure 2.2. Sample code snippet of a withdraw function.

In the provided example, any user can spoof any of the accounts the function iterates through. The main vulnerability is found in `wallet_info`, where a user can "fake" the account to point to the program's vault itself, essentially having the program withdraw from its funds and depositing the amount to the user.

The solution to the vulnerability in context would be to implement any sort of checks to find whether the `wallet_info` account is owned by the program itself, and not some outside entity. This can be accomplished through an `if` or an `assert` statement before withdrawing any actual funds, checking the condition of whether `wallet_info` is owned by the program as shown in Figure 2.3.

```
/// Using an assert statement
assert_eq!(wallet_info.owner, program_id);

/// Using an if statement
if wallet_info.owner != program_id {
    /// Return an error
}
```

Figure 2.3. Statements to add in verifying the integrity of `wallet_info`.

2.2 Missing Signer Checks

Smart contracts contain a different set of instructions that can be executed, similar to how different products can be selected from a vending machine. Some instructions may be sensitive and should only be called by certain accounts – for example, admin-related instructions should only be run by admins, and user-related instructions should only be run by users.

Within each account, a field `is_signer` is available as a flag to signify that the current account requires a signature for the transaction to execute. The signatures are keys to an account referenced to the instruction, notifying that the account has authorized the transaction [3].

The `withdraw` function from Figure 2.2 is still a great example of a sensitive instruction with no signer checks implemented at all. After patching up the ownership vulnerability, the accounts are now known to be data owned by the program. However, a user can spoof the `authority_info` account to execute an unauthorized, unsigned transaction to a sensitive instruction in withdrawing funds without the consent of another user.

Similar to the solution for missing ownership checks, a simple `if` or an `assert` statement checking the condition of whether `authority_info` has been signed is also a solution for missing signer checks, as seen in Figure 2.4.

```
/// Using an assert statement
assert!(authority_info.is_signer);

/// Using an if statement
if !authority_info.is_signer {
    /// Return an error
}
```

Figure 2.4. Statements to add in verifying the integrity of `authority_info`*.*

2.3 Signed Invocation of Programs

The Solana ecosystem supports programs that can call other programs through a mechanism called cross-program invocation, which is achieved through one program invoking an instruction of another program. One of the most common causes of invoking foreign programs is the use of the official Solana Program Library (SPL) for transferring funds to other accounts through its Token Program.

A function called `invoke_signed` from the Solana library is the instruction to achieve the feat – requiring an instruction input that consequently is supplied by an external user that automatically signs the provided account. If the original program has no checks validating the integrity of the program being invoked, a malicious program could be used unintentionally.

The correct way in invoking a program would be to validate the `key` field of the account holding the malicious program before invoking itself, verifying whether the program matches the program that is wanted to be invoked. An example piece of code can be seen below in Figure 2.5

for verifying whether SPL Token Program is, in fact, the program we want to run in transferring funds from one account to another.

```
fn withdraw(program_id: &Pubkey, accounts: &[AccountInfo], amount: u64)
  -> ProgramResult {
    let account_info_iter = &mut accounts.iter();
    let vault = next_account_info(account_info_iter)?;
    let vault_authority = next_account_info(account_info_iter)?;
    let destination = next_account_info(account_info_iter)?;
    let token_program = next_account_info(account_info_iter)?;
    . . .
    /// Using an assertion statement
    assert_eq!(token_program.key, &spl_token::id());

    /// Using an if statement
    if token_program.key != &spl_token::id() {
        return Err(ProgramError::InvalidTokenProgram);
    }

    invoke_signed(
        . . .
    )?;
    . . .
}
```

Figure 2.5. Sample code snippet of using `invoked_signed` *with verification.*

This scenario covers the use of the SPL Token Program. Fortunately, the newest versions of the SPL Token Program (3.1.1, specifically) include a validation statement relating to the program ID in their source code, so a check is not necessarily needed in Figure 2.5 when using up-to-date versions of dependencies such as the SPL. However, not all programs that a developer would want to invoke would include these checks explicitly, and numerous open-sourced projects still utilize older versions of the SPL Token Program.

2.4 Underflow and Overflow of Arithmetic Operations

One of the most general vulnerabilities that are found in any software application is the underflow and overflow of arithmetic operations, which are also found to be in Solana smart contracts. An arithmetic underflow occurs in an operation that results in a higher value, while an arithmetic overflow occurs in an operation that results in a lower value – this is mainly due to Rust wrapping around a numeric value using two's complement [4].

Within Rust and Solana, the underflow and overflow of arithmetic operations would mainly be found in programs compiled under *release* mode without the use of checked arithmetic operations [4]. When compiling programs under *debug* mode, the Rust compiler automatically inserts underflow and overflow checks for unchecked arithmetic operations, which do not accurately portray a program's source code.

Different examples can portray this vulnerability. MeanFi, an organization that strives to build decentralized applications as the financial equalizer for everyone, has a Decentralized Dollar Cost Averaging (DDCA) application. In short, the application allows individuals to automate investments without a centralized entity. It is open-sourced, and upon further examination, there is a function called `add_funds` that has an unchecked addition operation as shown below in Figure 2.6.

```rust
pub fn add_funds(
    ctx: Context<AddFundsInputAccounts>,
    deposit_amount: u64,
) -> ProgramResult {
    . . .
    /// Potential for an overflow (real code)
    ctx.accounts.ddca_account.total_deposits_amount += deposit_amount;
    /// A potential solution for overflow
    ctx.accounts.ddca_account.total_deposits_amount.checked_add(deposit_amount);
    . . .
}
```

Figure 2.6. Code snippet extracted from the `mean-core/ddca` *program.*

`add_funds` is an instruction that users can call when using the application. `deposit_amount` is a positive, numeric argument that users can provide to deposit an additional amount of funds to invest in the automated investment strategy. With an unchecked addition operation, the `total_deposits_amount` field has the potential to overflow, inaccurately portraying the funding of total deposits for a specific account. A solution to the vulnerability mainly involved the use of checked arithmetic functions, which is shown in Figure 2.6.

3. VERIFICATION TECHNIQUES

3.1 Static Analysis in Rust

The vulnerabilities aforementioned are generally known to developers in the Solana ecosystem, but much of the interest stems from having an automation tool that performs static analysis on different Solana programs to identify such vulnerabilities. In this section, we discuss several verification techniques that we were able to employ in a single tool to analyze programs for such vulnerabilities that have been deployed to the Solana blockchain.

Before discussing the various algorithms used to identify such vulnerabilities, some context of the static analysis tool we made should be considered to fully understand the design, as well as the methodologies and results that were obtained throughout this process. A brief outline of the architecture can be seen in Figure 3.1.

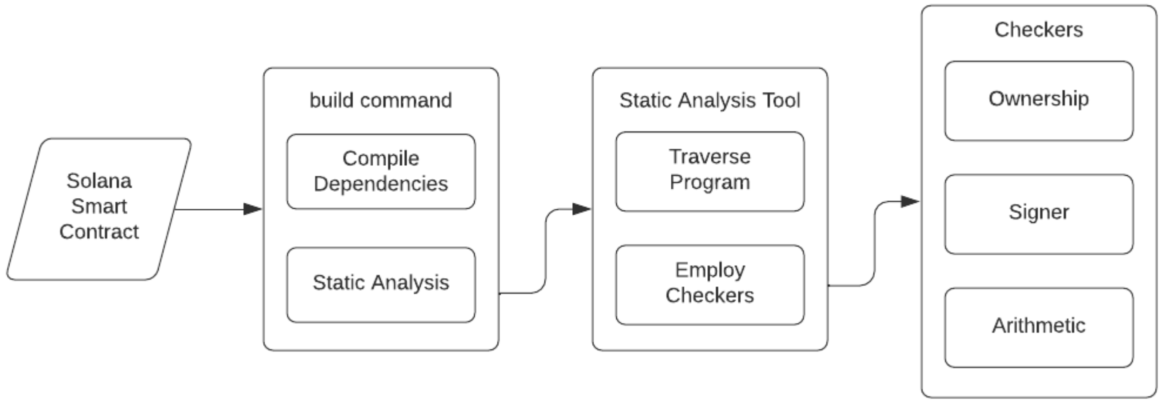

Figure 3.1. Outline of the process in running the static analysis tool.

A static analysis tool was implemented that integrates all the verification techniques in identifying the vulnerabilities mentioned using Rust. It utilizes data-flow analysis: an analysis that is concerned with the flow of data through a program. This includes the use of taint analysis, where the flow of external data is traced in the program to discover security vulnerabilities. A control-flow graph is used during the analysis of the program as a graphical representation of all paths in a program that are traversed through its execution, starting with the `entrypoint` function. To store states of the program such as tracking external data for taint analysis, we utilized a state machine during traversal.

In Rust, the tool encapsulates the use of `rustc` and its internal crates to generate a program's MIR (Mid-level Intermediate Representation). The MIR would then be analyzed by the static analysis tool with its verification techniques after traversing through the program as a control flow graph, simplifying the syntax of Rust programs and preserving type and debugging information. With the context of how the static analysis tool works, we now present the verification techniques for identifying the vulnerabilities that were introduced in Section 2.

3.2 Techniques to Identify Vulnerabilities

3.2.1 Ownership Checks

Ownership checks were made to traverse through the assertion and conditional statements of a smart contract's executable instructions. The technique is to validate whether a developer has inserted such statements that compare a variable with an `AccountInfo` type that accesses the `owner` field. Additionally, we check whether the field is comparing to a constant such as `spl_token::ID` or a parameter from the instruction such as `program_id` with `Pubkey`. Each function is verified through a system similar to taint analysis, where the flow of instructions can be regarded as safe, provided that they contain ownership checks.

```
fn processor::withdraw(_1: &Pubkey, _2: &[AccountInfo], _3: u64) -> Result<(),
  ProgramError> {

    . . .
    let _7: &solana_program::account_info::AccountInfo;
    bb44: {
        _93 = &((*_7).5: &solana_program::pubkey::Pubkey);
        _94 = &_1;
        (_92.0: &&solana_program::pubkey::Pubkey) = move _93;
        (_92.1: &&solana_program::pubkey::Pubkey) = move _94;
        _95 = (_92.0: &&solana_program::pubkey::Pubkey);
        _96 = (_92.1: &&solana_program::pubkey::Pubkey);
        _99 = _95;
        _100 = _96;
        _98 = <&Pubkey as PartialEq>::eq(move _99, move _100) -> bb45;
    }
    . . .

}
```

Figure 3.2. MIR representation of an assertion in ownership.

The assertion statement from Figure 2.3 can be represented in the MIR as shown in Figure 3.2. Assertion or conditional statements are represented through functions in `std::cmp::PartialEq::ne` and `std::cmp::PartialEq::eq`.

3.2.2 Signer Checks

Signer checks follow a similar algorithm to how we were able to detect any vulnerabilities relating to missing ownership checks. We would identify any assertion or conditional statements within the available instructions of a smart contract and determine whether the instruction contains any validations regarding signer checks. This involves checking variables that are of the `AccountInfo` type, along with the use of an `is_signer` field. To determine whether an instruction was sensitive to where it required a signed account, we utilized a standardized set of strings as a whitelist of variable names for `AccountInfo` to trigger a potential vulnerability, such as `authority_info`.

```
fn processor::withdraw(_1: &Pubkey, _2: &[AccountInfo], _3: u64) -> Result<(),
  ProgramError> {

    . . .
    let _23: &solana_program::account_info::AccountInfo;
    debug authority_info => _23;
    bb36: {
        _59 = ((*_23).1: bool);
        _58 = Not(move _59);
        switchInt(move _58) -> [false: bb38, otherwise: bb37];
    }
    bb37: {
        panic(const "assertion failed: authority_info.is_signer");
    }
    . . .

}
```

Figure 3.3. MIR representation of signer checks using assertions.

The MIR representation for an assertion statement in signer checks is slightly different from the ownership checks in Figure 3.2 when using the `assert!()` function. In Figure 3.3, `switchInt` is one of the instructions we identify for signer checks, in addition to the functions in `std::cmp::PartialEq` from how we handled ownership checks.

3.2.3 Signed Invocation of Programs

For vulnerabilities relating to the signed invocation of programs, we reported an issue when the smart contract signs an instruction to an outdated SPL Token Program in transferring funds to another account. This required validation of the smart contract's *Cargo.lock* file in verifying whether it utilized a version of the SPL Token Program older than 3.1.1 – if it was newer, we do not continue to check for specific instructions relating to a signed invocation in transferring funds.

However, once we identify that the smart contract could be vulnerable with an older version of the SPL Token Program, the smart contract is traversed to identify any instructions that call any function from `spl_token::instruction`. Functions that did contain the instructions would use a procedure similar to taint analysis in tracing through the flow of instructions that validates whether the inputted program ID and its instructions are checked – when there are no checks, we flagged such functions as potential vulnerabilities.

```
fn processor::withdraw(_1: &Pubkey, _2: &[AccountInfo], _3: u64) -> Result<(),
  ProgramError> {

    . . .
    bb58: {
        . . .
        _135 = transfer_checked(move _136, move _138, move _140, move _141, move
               _142, move _143, move _146, move _147) -> bb59;
    }
    bb64: {
        . . .
        _131 = invoke_signed(move _132, move _148, move _160) -> bb65;
    }
    . . .

}
```

Figure 3.4. MIR representation of cross-invocation in SPL Token Program.

The validation of the inputted program ID and its instructions are done in a similar algorithm to the ownership checks in Section 3.2.1, where we verify that the official `spl_token::ID` matches the inputted program ID through conditional statements. However, we first identify any cross-invocation of SPL Token Programs through the MIR similar to Figure 3.4. Extra debug information shows that a function `spl_token::instruction::transfer_checked` would be invoked.

3.2.4 Underflow and Overflow of Arithmetic Operations

Lastly, we identified arithmetic underflow and overflow through the flow of external data. When an external argument was utilized in any arithmetic operations throughout the smart contract, and if the operation was an unchecked arithmetic operation that could underflow or overflow, we would report the issue.

```
fn ddca::add_funds(_1: anchor_lang::Context<AddFundsInputAccounts>, _2: u64) ->
  std::result::Result<(), anchor_lang::prelude::ProgramError> {

    . . .
    debug ctx => _1;
    debug deposit_amount => _2;
    _118 = _2;
    _119 = <anchor_lang::Account<DdcaAccount> as DerefMut>::deref_mut(move _120)
           -> bb50;
    bb50: {
        ((*_119).9: u64) = Add(((*_119).9: u64), move _118);
    }
    . . .

}
```

Figure 3.5. MIR representation of an unchecked addition operation.

Within the Rust MIR, an unchecked arithmetic operation would be any operations that do not use any of the checked functions, such as a `checked_add()`. These operations fall under an `Rvalue::BinaryOp` if they are unchecked. The MIR representation of Figure 2.6 from the DDCA program is seen in Figure 3.5, with the `Add()` function being an unchecked arithmetic operation with an external parameter in `deposit_amount`.

4. METHODS

An automated static analysis tool was utilized to combine all the aforementioned techniques to identify vulnerabilities in real-world applications built with Solana. To apply the tool to the applications, two methods were conducted to find the results. One, we developed scripts that were run daily to use the tool in a fixed state on all the programs provided on the GitHub repo from the Solana Program Library as it progressed through commits. Two, we used similar scripts that were also run daily to use the tool as we improved its reliability on a fixed state of programs.

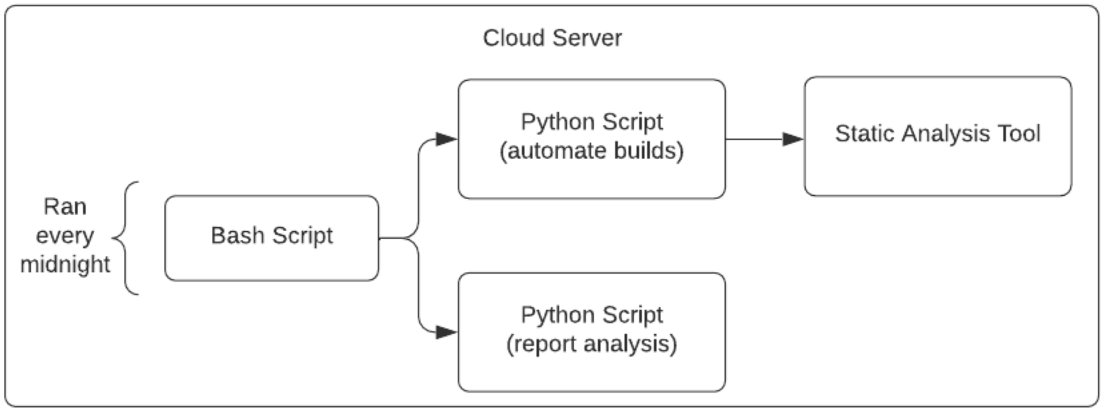
Cloud Server
Ran every midnight
Bash Script
Python Script (automate builds)
Static Analysis Tool
Python Script (report analysis)

Figure 4.1. Outline of the structure in conducting the experiments for Sections 4.1 and 4.2.

As shown in Figure 4.1, the scripts were a combination of Bash and Python scripts that automated the process of pulling GitHub commits, building and compiling the tool and the tested programs, and generating a report of vulnerabilities. A cloud server was utilized with a version of Ubuntu 20.04.3 LTS provided by Amazon Web Services and a CronJob scheduled every midnight to run such scripts. More details of the experiments are explained in the subsequent sections.

4.1 Fixed Tool on Updated Programs

An experiment on the static analysis tool was conducted where the tool and verification techniques were fixed in a state where no updates were given to improve. In this state, we ran the tool on 30+ programs that the tool identified from the Solana Program Library, which was open-sourced and published on GitHub with changes constantly committed every day.

A Python script was developed to automate the process of building the static analysis tool, discovering a set of programs from a specific source directory, and running the static analysis tool on all the programs with generated reports of specific vulnerabilities that were found. Additionally, another Python script was created to discover any changes in the vulnerabilities between two specific days, mainly concerned with the reports that were generated from the previous and current day.

To help run the scripts automatically and consistently at midnight, commands to run the Python scripts were encapsulated into a single Bash script that was called by the cloud server from a CronJob. This method was running for two weeks, starting in March, with the results manually inspected thereafter.

4.2 Updated Tool on Fixed Programs

In addition to having a constant, fixed static analysis tool running on a constantly changing set of Solana programs, we were able to conduct an experiment with the tool and verification techniques that was constantly updated to improve the rate of false positives and the identification of vulnerabilities that were not previously detected. The tool would go through changes mostly every day and would be used to run a set of fixed programs that we found through open-sourced means.

Bash and Python scripts were used similarly to the experiment above, with minor differences to match the constraints of the experiments. A CronJob was made to call the scripts by the cloud server every midnight, and the experiment was running since having a stable version of the tool in February.

The programs we compiled were from GitHub that were successful and stable builds. There was a total of over 30 programs that the tool was able to identify, with some of them being dependencies. Additionally, we selected the latest versions of these programs, with several deployed to the Solana blockchain.

4.3 Initial Testing on Fixed Programs

Lastly, we used several noteworthy applications with an active userbase, entirely unrelated to the programs and procedures from the former experiments when we initially built the static analysis tool. These applications are open-sourced and were obtained from the `mean-dao/mean-core`, `blockworks-foundation/mango`, and `metaplex-foundation/metaplex-program-library` repos on GitHub.

There were no automation procedures when conducting this experiment; rather, we used the mentioned applications, in addition to the minimal applications we made explicitly containing vulnerabilities, as a baseline for identifying vulnerabilities when creating the static analysis tool during the early stages and verifying the correctness of the tool itself.

5. RESULTS

5.1 Results from Initial Testing on Fixed Programs

As mentioned in Section 4.3, we ran the static analysis tool on a set of programs after integrating the verification techniques into a program that was stable enough to run with the results shown in Table 5.1. These programs were obtained with the latest commits from GitHub in late January 2022.

Table 5.1. Initial results from the static analysis tool on fixed programs.

	Ownership Checks	Signer Checks	Signed Invocation	Arithmetic Operations	Total
`mean-dao/mean-core/ddca`	0	0	0	2	2
`metaplex-foundation/metaplex-program-library/metaplex`	10	0	0	10	20
`blockworks-foundation/mango`	0	0	0	2	2

For the DDCA program, there were a total of 2 vulnerabilities found from the static analysis tool. One of the issues was considered a false positive from utilizing the Anchor framework in building the program, as it was a private helper function that could not be accessed by an attacker. However, there was another integer overflow issue that was problematic from what the tool reported, which was shown in Figure 2.6 under the discussion of identifying and solving such issues.

Moreover, the main Metaplex program contained 20 total vulnerabilities. There were 10 cases where there were missing ownership checks; however, upon further inspection, all of them were found to be false positives, with the functions calling a separate `assert_owned_by` utility function with ownership assertions before handling sensitive instructions. Additionally, there were 10 issues for unchecked arithmetic operations, which were found to be true, but were concerns that could not be manipulated as they were either private helper functions or functions that checked the validity of the inputs before performing any unchecked arithmetic operations. Overall, nothing significant was found from the vulnerabilities that were detected through static analysis.

Lastly, the Mango Markets program was found to have 2 total vulnerabilities. Both were found to be unchecked arithmetic operations pointing to the same line of code but approached through a different set of call stacks. Unfortunately, nothing of significance was found from this line of code, as it was a function handling the state of the program in invoking arithmetic operations related to time, with heavy validation on the inputs.

Two of the three programs that were initially run to test the static analysis tool are known to use the Anchor framework. The main Metaplex program was the lone project to not use the framework. This framework allows developers to conveniently write smart contracts, with several provisions in the framework that implements checks in ownership and signatures automatically, unlike developing a smart contract from scratch. Additionally, all three programs utilized the latest versions of dependencies, such as the SPL Token program, to where signed invocation vulnerabilities could not exist. Thus, the results shown in Table 3.1 are somewhat reliable in this initial prototype of the static analysis tool after further investigation.

5.2 Fixed Tool on Updated Programs

A total of 38 programs from the Solana Program Library were identified from the GitHub repo by the static analysis tool, and an analysis of the two significant programs in the Token and Token 2022 programs was conducted for two weeks between March 2nd and March 17th from changes committed by the community. The results are shown in Figure 5.1.

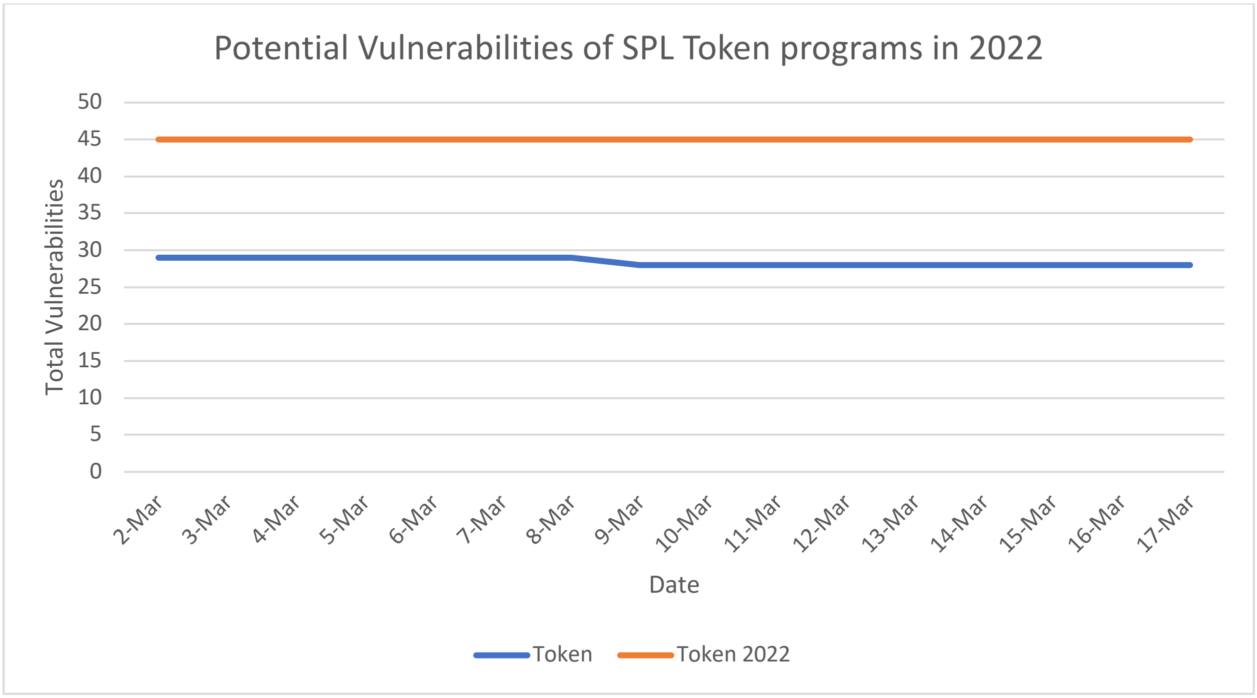

Figure 5.1. Graph of total vulnerabilities over time from the SPL Token programs.

A total of 22 commits were made throughout the period, with most of the commits having some involvement with the Token programs that were being investigated. No significant results were found, as the vulnerabilities that were discovered with the programs were relatively constant except for a commit that reduced a vulnerability in the Token program on March 8th.

The regular Token program had an initial total of 29 potential vulnerabilities. One of the vulnerabilities was an integer overflow issue with an unchecked addition operation, but the issue could not be used for malicious intent as the underlying functionality was for formatting numbers as strings. All the other vulnerabilities were mainly involved with the missing owner and signer checks, which were also found to be insignificant as they call the program's `check_account_owner` or `cmp_pubkeys` functions, which subsequently calls a `sol_memcmp` system function in comparing `Pubkeys`. The one reduced vulnerability after March 8th came from altering code relating to ownership checks, with more assertions added in verifying the input accounts from the flagged function.

The newer Token 2022 program had a constant number of vulnerabilities detected, even with commits being made throughout the source code. 45 total issues were found, with 20 relating to arithmetic operations and 25 relating to missing ownership and signer checks. The arithmetic operations noted also were unchecked for potential underflow or overflow issues, but the concerns were not significant for similar reasons in the regular program – there was no purpose in utilizing these operations for malicious intent, with most of the operations occurring within private helper functions. Additionally, the Token 2022 program has utility functions in `check_account_owner` or `cmp_pubkeys` that are called before sensitive operations to validate ownership and signed instructions, which covered all the missing check vulnerabilities.

To summarize, the vulnerabilities that were detected in two weeks were mainly insignificant, but the static analysis tool was able to identify sensitive instructions and arithmetic operations that needed various checks on programs. Even though the programs we investigated were mature, actively maintained, and heavily audited, the verification techniques employed in these programs convey that it could serve as a baseline to identify basic security issues.

5.3 Updated Tool on Fixed Programs

There was a total of 31 programs we compiled into a dataset that the static analysis tool found. The total number of vulnerabilities found between February 4th and March 21st for the notable programs from the dataset is shown in Figure 5.2.

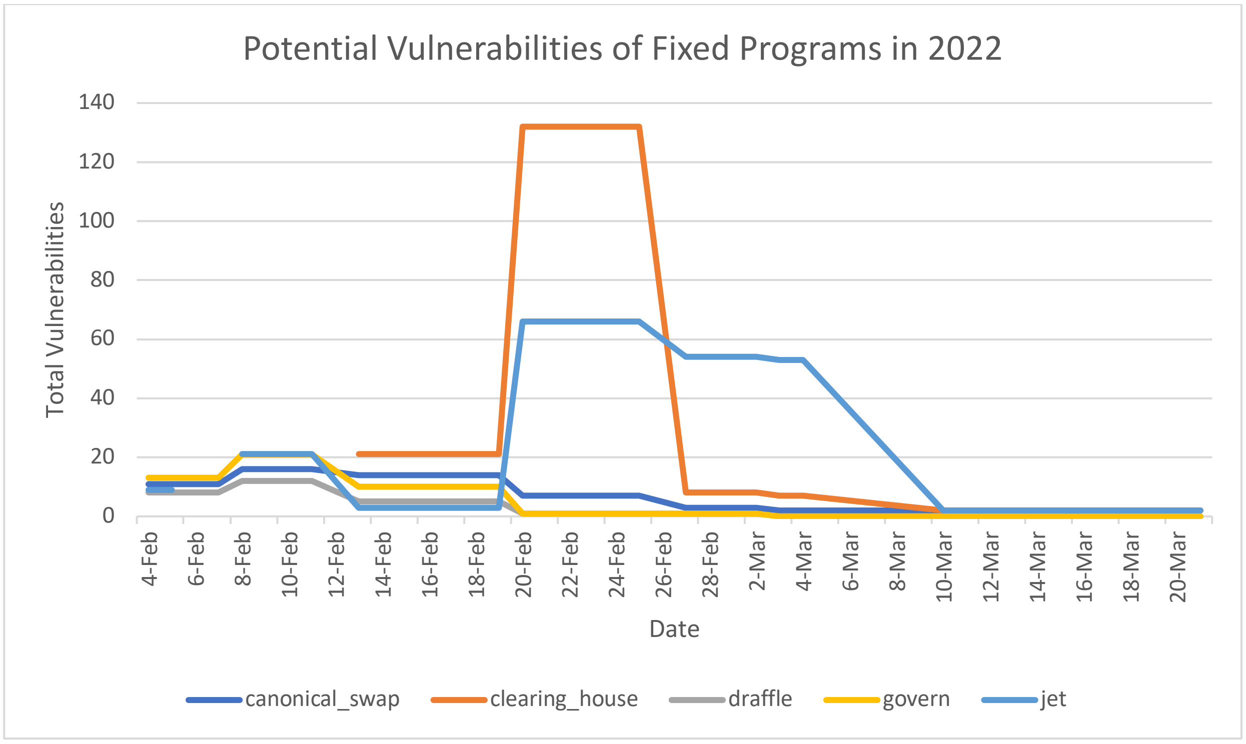

Figure 5.2. Graph of total vulnerabilities over time from the fixed programs.

Several trends relating to the total number of vulnerabilities occurred as we modified the verification techniques. On February 19th, we deployed several changes to the algorithms relating to arithmetic operations to fit in a generic taint analysis interface like the other techniques, increasing the rate of false positives on the issue. An additional filtering mechanism was added afterward on February 25th, removing any duplicate reports and blacklisting a number of functions that created false positives. Throughout the process, heuristics were tweaked for the other vulnerabilities but there were no significant impacts that were seen similar to the issues related to arithmetic operations.

The initial results contained around 10 false positives per program, where they were mainly issues of arithmetic operations and signer checks. Over time, we improved the overall reliability of the verification techniques to where there are few false positives in the end. The current issues now are mainly related to false positives in arithmetic operations.

5.4 General Performance

Throughout the experiments, performance remained constant in terms of the time it took to run the analysis and the reliability of the reports produced by the tool. Each smart contract took an average of 4 minutes for the static analysis tool to generate a report on any vulnerabilities it discovered. This was consistent with all the programs that we used in our testing environment outlined in Section 4. Most of the time comes from the compilation process that is conducted by the Rust compiler, as it loads crates and dependencies from programs before allowing our tool to run static analysis.

Additionally, the static analysis tool was reliable enough in finding clear vulnerabilities with the techniques we mentioned earlier. The identification of sensitive instructions and the reporting of potential vulnerabilities were sufficient in this early stage of development to where the tool should be used in practice for Solana developers in securing their applications.

6. CONCLUSION

In this paper, we have presented several different verification techniques to identify vulnerabilities in Solana smart contracts. The vulnerabilities were missing ownership checks, missing signer checks, the signed invocation of programs, and the underflow and overflow of arithmetic operations. In conjunction with a static analysis tool that combined these techniques, we were able to showcase the results we found from the development of this tool on real-world applications.

There is future work to be accomplished. We wish to improve the reliability in identifying more true positives and reduce the number of false positives in vulnerabilities. Improving the reliability of the tool will allow us to implement more techniques in identifying new vulnerabilities. Additionally, to verify and reproduce true positive vulnerabilities, we would create an automated tool in generating a proof-of-concept program that exploits the smart contracts that were tested on.

We believe that the techniques discussed within the paper have the potential to be incorporated into a single static analysis tool for improving the security of Solana programs. In various experiments, we found that the verification techniques were able to identify sensitive instructions and properly raise concerns about potential vulnerabilities that needed attention. Early versions of the tool have shown promise in identifying major security concerns of a program before deployment to where we can safely encourage Solana developers to secure their applications with the techniques shown in practice.

REFERENCES

[1] Vitalik Buterin. 2014. *Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform*. Retrieved January 16, 2022 from https://ethereum.org/669c9e2e2027310b6b3cdce6e1c52962/Ethereum_White_Paper_-_Buterin_2014.pdf

[2] Anatoly Yakovenko. *Solana: A new architecture for a high performance blockchain v0.8.13*. Retrieved January 16, 2022 from https://solana.com/solana-whitepaper.pdf

[3] Solana. 2022. *Accounts*. Retrieved January 31, 2022 from https://docs.solana.com/developing/programming-model/accounts

[4] The Rust Programming Language. *Data Types*. Retrieved January 19, 2022 from https://doc.rust-lang.org/book/ch03-02-data-types.html

[5] Solana. 2022. *Deploying*. Retrieved January 31, 2022 from https://docs.solana.com/developing/on-chain-programs/deploying

[6] Nick Szabo. 1997. *The Idea of Smart Contracts*. Retrieved February 27, 2022 from https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/idea.html