VRust: Automated Vulnerability Detection for Solana Smart Contracts

Siwei Cui
siweicui@tamu.edu
Texas A&M University
College Station, Texas, USA

Gang Zhao
zhaogang92@tamu.edu
Texas A&M University
College Station, Texas, USA

Yifei Gao
keizoku-pravda@tamu.edu
Texas A&M University
College Station, Texas, USA

Tien Tavu
tientavu@tamu.edu
Texas A&M University
College Station, Texas, USA

Jeff Huang
jeff@cse.tamu.edu
Texas A&M University
College Station, Texas, USA

ABSTRACT

Solana is a rapidly-growing high-performance blockchain powered by a Proof of History (PoH) consensus mechanism and a novel stateless programming model that decouples code from data. With parallel execution on the PoH Sealevel runtime (instead of PoW), it achieves 100X-1000X speedups compared to Ethereum in terms of transactions per second. With the new programming model, new constraints (owner, signer, keys, bump seeds) and vulnerabilities (missing checks, overflows, type confusion, etc.) must be carefully verified to ensure the security of Solana smart contracts.

This paper proposes VRust, the first automated smart contract vulnerability detection framework for Solana. A key technical novelty is a set of static analysis rules for validating untrustful input accounts that are unique in the Solana programming model. We have developed a total of eight different vulnerability types, and VRust is able to check all of them fully automatically by translating source code into Rust MIR-based inference rules without any code annotations. VRust has been evaluated on over a hundred of Solana projects, and it has revealed 12 previously unknown vulnerabilities, including 3 critical vulnerabilities in the official Solana Programming Library confirmed by core developers.

CCS CONCEPTS

• **Software and its engineering → Correctness**; **Software safety**; • **Security and privacy → Distributed systems security**; **Domain-specific security and privacy architectures**.

KEYWORDS

Blockchain Security, Solana, Verification

ACM Reference Format:
Siwei Cui, Gang Zhao, Yifei Gao, Tien Tavu, and Jeff Huang. 2022. VRust: Automated Vulnerability Detection for Solana Smart Contracts. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22), November 7–11, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3548606.3560552

1 INTRODUCTION

Smart contracts are one of the most appealing features of blockchain technology [1]. They have grown in popularity as they facilitate the need for distributed, transparent, and trustless computing. With more than $1TB of market capitalization [60], cryptocurrencies driven by smart contracts are attractive targets for attackers because vast amounts of funds could be stolen directly by exploiting vulnerabilities in open-source code. For example, the Wormhole network [26, 27] lost more than $320 million due to a missing key check. Another protocol, Poly Network, lost $611 million due to a vulnerability in its smart contract [40].

Ethereum (ETH) [16] is among the most well-known smart contract platforms where thousands of applications have been deployed with more than $374 billion in market capitalization [13]. However, Ethereum has limited its processing speed to only 10-15 TPS (transactions per second) [30]. Besides the use of PoW, another reason is that every Ethereum smart contract maintains a single copy of states, updating the states to the same smart contract has to run sequentially.

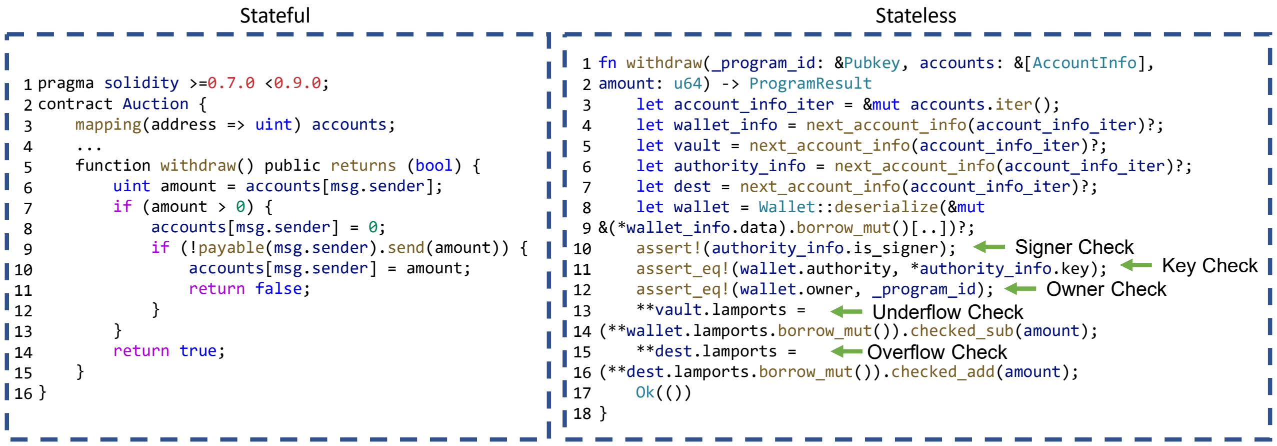
Solana [57] is a rising smart contract platform designed to resolve ETH's speed limit. It is considered to be the world's fastest blockchain, with transaction speeds of over 2K TPS [46]. With Solana's scalability, transaction fees are under $0.01, and transaction processing rates can reach 400 milliseconds per block [56]. It allows previously underserved consumers to access the decentralized ecosystem in an efficient manner, including Decentralized Finance (DeFi) [11], Non-Fungible Tokens (NFTs) [12], and other Web 3.0 applications [25].

Table 1 compares the key differences between Ethereum and Solana. The stateless programming model and consensus mechanism contribute to the fast transaction speed of Solana over Ethereum.

Ethereum adopts a stateful mechanism where all network transactions are stored in a single state. Therefore, whenever a new transaction happens, the whole network must update all participants' copies of the states to avoid double-spending or contradictory states. Solana incorporates a stateless programming model in which instructions define which data they will alter

Table 1: A Comparison Between Ethereum and Solana (Statistics as of Apr 05 2022).

Smart Contract	Ethereum (ETH 1.0)	Solana
Programming Model	Stateful: logic and state are combined	Stateless: code and data are decoupled
Programming Language	Solidity	Rust, C, and C++
Consensus Mechanism	PoW and PoS (ETH 2.0)	PoH
Launch Date	July 2015	April 2019
Scalability	15 TPS	2,000 TPS
Average Transaction Fee	$15	$0.00025



```
pragma solidity >=0.7.0 <0.9.0;
contract Auction {
    mapping(address => uint) accounts;
    ...
    function withdraw() public returns (bool) {
        uint amount = accounts[msg.sender];
        if (amount > 0) {
            accounts[msg.sender] = 0;
            if (!payable(msg.sender).send(amount)) {
                accounts[msg.sender] = amount;
                return false;
            }
        }
        return true;
    }
}
```

```
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 vault = next_account_info(account_info_iter)?;
    let authority_info = next_account_info(account_info_iter)?;
    let dest = next_account_info(account_info_iter)?;
    let wallet = Wallet::deserialize(&mut
&(*wallet_info.data).borrow_mut()[..])?;
    assert!(authority_info.is_signer);
    assert_eq!(wallet.authority, *authority_info.key);
    assert_eq!(wallet.owner, _program_id);
    **vault.lamports =
(**wallet.lamports.borrow_mut()).checked_sub(amount);
    **dest.lamports =
(**dest.lamports.borrow_mut()).checked_add(amount);
    Ok(())
}
```

(a) Ethereum Model -- Solidity (b) Solana Model -- Rust

Figure 1: A Key Difference Between Ethereum and Solana Programming Models.

in advance. Each transaction does not require changing the network's whole state and can be carried out in arbitrary order. The Solana runtime organizes incoming transactions such that non-overlapping transactions can be processed simultaneously.

To illustrate stateful vs stateless programming, we compare a `withdraw` contract[1] written for Ethereum in Solidity and the same function written for Solana in Rust in Fig. 1. In Ethereum, all states used (read or modified) by the contract are a part of the contract, in a way similar to class variables declared in object-oriented programming. In contrast, in Solana, all states including accounts must be supplied as input parameters, similar to data pointers. In this way, Solana runtime can run multiple copies of a smart contract in parallel on different input accounts.

The stateless design makes Solana smart contracts more complicated to write compared to Ethereum. All inputs are provided externally, which could be faked by attackers and must be carefully validated. In particular, several new types of programming constraints such as owner, signer, and key validation checks are unique to Solana smart contracts. If any of the validation checks is missed in a smart contract, an attacker may provide a fake input to exploit it. The Wormhole attack is a prominent example.

We propose VRust, a novel end-to-end static analysis framework that can automatically detect most common security vulnerabilities in terms of eight patterns of common pitfalls in Solana smart contracts. VRust checks all these patterns fully automatically by translating Rust source code into Mid-level Intermediate Representation (MIR) and constructing novel inference rules based on MIR and inter-procedural data flow analysis without any code annotations.

Our contributions are summarized as follows.

- To the best of our knowledge, VRust is the first automated vulnerability detector for Solana smart contracts.
- We designed a novel end-to-end approach that incorporates whole program static data flow analysis on Rust MIR-based inference rules, which can be adapted to detect multiple vulnerability patterns.
- Our prototype tool, VRust, has been evaluated on over a hundred open source Solana projects, and has revealed 12 new vulnerabilities, including 3 critical vulnerabilities in the official Solana Programming Library confirmed by their core developers.

The rest of this paper is organized as follows. Section 2 introduces the background knowledge that is essential for Solana smart contracts. Section 3 presents the design of our end-to-end static analysis tool, and Section 4 describes implementation details. In Section 5, we evaluate VRust on 12 real-world applications built on top of Solana smart contracts, measuring efficiency and performance. Section 6 evaluates the limitations of VRust. We present related work in Section 7, and conclude in Section 8.

[1]The Ethereum example is simplified from https://soliditylang.org/ and the Solana example is abstracted from https://github.com/neodyme-labs/neodyme-breakpoint-workshop.

```rust
pub fn verify_signatures(
    ctx: &ExecutionContext,
    accs: &mut VerifySignatures,
    data: VerifySignaturesData,
) -> Result<()> {
    ...
    let current_instruction =
    solana_program::sysvar::instructions::
        load_current_index(
        &accs.instruction_acc.try_borrow_mut_data()?,
    );
```

Listing 1: Missing Key Check in Solana Wormhole Network.

2 BACKGROUND

2.1 Wormhole Vulnerability

One of the motivating cases leading to the development of VRust was the exploit found in the Wormhole network that caused a loss of more than $320 million [26, 27]. The Wormhole network facilitates transactions between blockchain platforms before transferring a token to another chain. They validate signatures of the guardians to ensure safety. However, the input sysvar (synthesized data accounts for network states) program account key was not checked (Lst. 1) to validate that it is the valid sysvar account by the provided function. In Wormhole, sysvar (formally instruction_sysvar) is used to store a set of signatures produced by a function verify_signatures. Without verifying the validity of the sysvar program account, attackers could generate a fake instruction sysvar account with malicious data, and the signatures could be forged using previously valid transferred tokens. As a result, all signatures in the signature set can be falsely set to true, indicating that it contains all valid signatures. Hackers could invoke of an unauthorized mint to their accounts.

The root cause is a flaw in the Wormhole bridge code base for verifying signatures. It used an unsafe and deprecated Solana API to parse the input accounts without validating the account keys. This vulnerability can be fixed by either adding a key check to the account (Lst. 5), or using an updated Solana API in `load_current_index_checked` (Lst. 6). We discuss more details in Section 5.1.

2.2 Solana Programming Model

When an application communicates with a Solana cluster, it sends transactions containing one or more instructions [48]. Each instruction specifies a single program (smart contract) with accounts and a data byte array passed to it. The program interprets the data array and executes the instructions on the provided accounts. The Solana runtime forwards those instructions to the applications that have already been deployed. For each transaction, instructions are performed sequentially and atomically. All account modifications in the transaction are discarded if any of the instructions is incorrect.

Several terminologies are crucial to understanding of the Solana programing model.

- *Program ID.* Public key of an account containing programs.
- *Account.* A record in Solana that either contains data or a program (smart contract) to be executed. A Solana account contains a currency known as lamports. Solana identifies an account with a public key. With a stateless programming model, a list of accounts is supplied as function arguments `accounts` at Line 1 in Fig. 1 (b). Accounts (e.g., `wallet_info`, `authority_info`) are decoupled by an iterator `accounts.iter()` at Line 3.

Fig. 2 illustrates the structure of a Solana account. The `lamports` and `data` fields in the account can be updated for each account by its account owner. Other fields (key, `is_signer`, etc.) can only be managed by the Solana system program and are not writable by user programs.

- *Account Owner.* The address of the program that owns the account. The account's data can only be edited by the account's owner. In Fig. 1 (b), an owner is checked at Line 12.
- *Signer.* A signer is an account accompanied by a digital signature in the account's metadata to indicate that the account owner has authorized the transaction. In Fig. 1 (b), a signer is checked at Line 10.
- *SOL and Lamports.* SOL is the native token of a Solana smart contract. A lamport is a fractional native token with the value of 0.000000001 SOL.
- *Solana Program Library (SPL).* The SPL is a set of on-chain applications designed to facilitate tasks such as creating tokens (digitally transferable assets).
- *Wallet.* A wallet is a set of key pairs, providing a way for users to manage their funds. The Public key (Pubkey) is known as the wallet's receiving address.
- *Program Derived Address (PDA).* A PDA is a 32-byte string in the format of a public key, but without a corresponding private key [49]. Generated from a combination of seeds and a program id, PDAs are used for cross-program invocation.
- *Bump Seeds.* 8-bit integers served as additional seeds and appended to PDA seeds. Bump Seeds are used to generate PDAs satisfying the ed25519 constraint [8] specific to Solana.

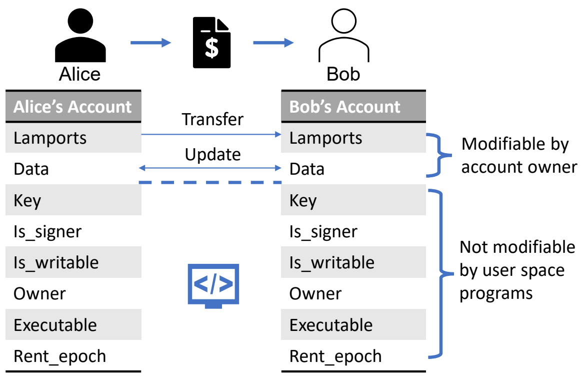


Figure 2: Solana Account Design.

2.3 Solana vs. Ethereum: Design Philosophy

We compare the execution model for Solana and Ethereum in Fig. 3. The most significant difference between Ethereum and Solana is

Table 2: Bug Patterns for the VRust Framework.

VRust Bug Patterns	Description
Missing Signer Check	The is_signer field in AccountInfo demonstrates that a user signs an instruction to be executed.
Missing Owner Check	The owner field in AccountInfo indicates whether a supplied account is owned by the program.
Missing Key Check	The key needs to be verified when accessing the data field of an account.
Overflow / Underflow	An arithmetic overflow/underflow can misrepresent numerical data.
Account Confusion	The account data structure can be misinterpreted with another structure of the same data length.
Cross Program Invocation	Programs can invoke instructions from other programs through cross-program invocation.
Numerical Precision Error	Precision loss from rounding operations in transactions can misrepresent numerical data.
Bump Seeds	Bump seeds of program addresses should be verified to prevent tampering with data.

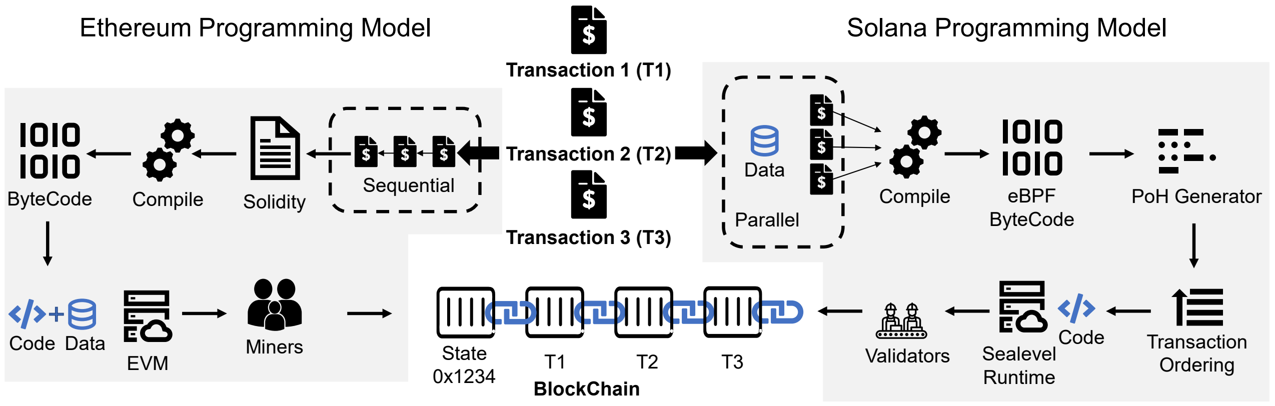


Figure 3: A Comparison Between Solana and Ethereum on Executing Transactions.

the stateful/stateless programming model, i.e., how code and data are represented in smart contracts. In Ethereum, data and code are coupled together. A smart contract comprises both the code and the data (states). In Fig. 1 (a), the state variable `accounts` (line 3) is data, and the function `withdraw` is code. Ethereum is stateful as it stores key data as states of the smart contract on the blockchain. Each transaction requires a change of one or more states in the ETH network, as the data (states) are maintained in the ETH smart contract. This design of coupling code and data is intuitive, but it limits the performance and scalability. Whenever a new transaction happens, the whole network must update all its local copies to reflect the new state changes.

Solana introduces a new stateless programming model where data and code are decoupled. All data (states) a smart contract reads or writes must be passed as inputs, including user accounts, thus Solana requires validating all input accounts. Solana's parallel runtime model, Sealevel [58], decouples data (e.g., account balances) from code [55]. Instructions in Solana define which data they will alter in advance. In Fig. 1 (b), the function `withdraw` is the code, and the state variables `_program_id`, `accounts`, and `amount` are the data supplied as inputs. This new programming model involves the whole life cycle of the Solana blockchain, including the blockchain setup and all transaction executions.

With this decoupling design, transactions from different accounts invoking the same Solana smart contract can run in parallel, achieving high scalability. The Solana VM organizes incoming transactions such that non-overlapping transactions can be processed simultaneously. Solana's capacity to execute a block every 400 ms and record 60k TPS (at peak period) is unquestionably a key selling point [5]. Tower Byzantine Fault Tolerance [51], an improvised form of the practical Byzantine Fault Tolerance Algorithm [10], is adopted by Solana to improve efficiency. The stateless design in Solana is the most essential difference compared to Ethereum, in which user accounts are stored as heap variables used directly in the smart contracts.

However, the stateless programming model also increases the risk of untrusted input and raises security issues exclusively to Solana. Since all accounts are provided as inputs when invoking a Solana program, users can supply arbitrary accounts and there's no built-in mechanisms stopping a malicious user from doing so with fake data. With the stateless programming model, Solana opens new attack surfaces that do not exist in Ethereum, because input accounts may not be trusted. Solana programs must check the validity of the input accounts properly.

There are also many other key differences between Ethereum and Solana in terms of vulnerability detection. One example is the bump seeds vulnerability, which is unique to Solana. PDAs of a Solana program could have multiple valid bumps with the same seeds, giving multiple different addresses. Therefore, PDAs can be faked, and a smart contract should validate the bump seeds before executing critical transactions. Another example is that Solana only allows self-recursion at a fixed depth due to the restriction of cross-program invocation, avoiding most reentrancy bugs in

Ethereum [47]. Those differences bring new challenges towards vulnerability detection on Solana.

2.4 Hyperledger Fabric

Hyperledger Fabric [3] is an open-source blockchain framework for distributed ledger solutions [17]. It is hosted by the Linux foundation's Hyperledger project proposed in 2015, and is private and permissioned. User identities are known and authenticated in Fabric, which is different from the case in the Solana blockchain. Fabric's modular architecture allows applications implemented with high confidentiality, resiliency, and pluggable components of complex economic systems. [29]

Fabric operates on different kinds of assets, which could be anything with monetary value from real estate to intellectual properties [18]. Assets are created and modified by smart contract code (known as chaincode) [6], which handles business logic and defines rules of reading and writing key-value pairs stored in the state database. By executing chaincode, state updates are submitted to the Fabric network and applied to all copies among the peers. Different from the stateless design in Solana, the states created by a chaincode in Fabric are scoped exclusively within that chaincode and cannot be accessed directly by other chaincodes [28].

The state database in Fabric is stored as a channel ledger, containing sequenced tamper-resistant state transition records on chain. Although Fabric supports parallel transaction execution, the states are still on the chain, and transactions modifying the states are required to be executed sequentially.

Fabric emphasizes data privacy through the design of channels and the permissioned provided by data access control. Different from Solana which is a public, open-source and decentralized platform featuring transparency, Fabric consists of multiple different channels to provide privacy. Each channel in Fabric can define a chaincode of its own, and every channel maintains one shared ledger among all its peers. Similar to Ethereum, ledger data is also stored within the scope of a channel on-chain.

3 TECHNICAL APPROACH

The VRust framework has been developed to detect eight types of vulnerabilities, as shown in Tab. 2. Our framework is depicted in Fig. 4.

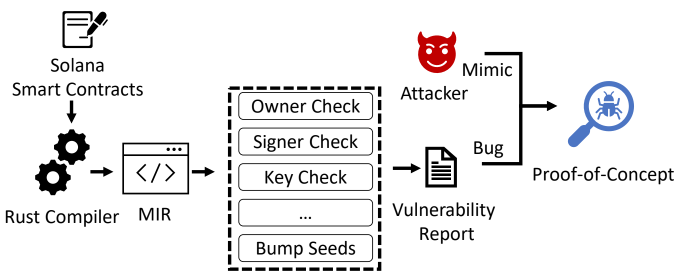


Figure 4: An Overview of the VRust Framework.

3.1 VRust Checker

We focus on a subset of Rust MIR [44] instructions, called `VRustMIR` (Lst. 2), that covers all the necessities for vulnerability detection on the Solana blockchain.

```
VRustMIR = fn({TYPE}) -> TYPE {
    {let [mut] B: TYPE;}
    {let TEMP: TYPE;}
    {BASIC_BLOCK}
};
BASIC_BLOCK = BB: {STATEMENT} TERMINATOR
STATEMENT = LVALUE "=" RVALUE

TERMINATOR = SWITCH(LVALUE, BB...)
           | CALL(LVALUE0 = LVALUE1(LVALUE2...), BB0,
           ↪  BB1)

LVALUE = TEMP
       | ARG
       | RETURN

RVALUE = Use(LVALUE)
       | CONSTANT
       | LVALUE as TYPE
       | LVALUE <BINOP> LVALUE
       | Struct { f: LVALUE0, ... }
```

Listing 2: VRustMIR Definition.

We propose a general fixed-point data (state) flow analysis on the Rust MIR to reason about vulnerabilities in Solana smart contracts. The Control Flow Graph (CFG) of a Solana smart contract has an entry point function called `fn entrypoint(_1: *mut u8) -> u64` that dispatches target public function APIs. We design a novel state machine for conducting interprocedural analysis on Solana smart contracts that are extensible and efficient. On a high level, our whole program analysis starts with an entry point, builds the initial state S and call stack CS, and updates the state when we encounter different statements.

VRust traverses every direct or indirect function that can be resolved locally. In Solana projects, loops are rarely seen in the real-world codebase. We implemented a fixed-point algorithm by traversing the basic blocks in reverse postorder so that the state would be stable after one iteration in most cases.

Based on the design of our state flow analysis, we have implemented and developed eight checkers. For each checker, all that is required is to set up the initial state and design the state transition function. It minimizes the efforts to implement new checkers for previously unseen patterns.

Our VRust framework can easily be extended to perform other sets of analyses. The state machine design helps simplify the workload to introduce checkers on new statements. For each statement, all we need is to implement the update function for both pre-state and post-state. To extend the analysis, one should define a custom state structure and implement the StateTransitor trait, in which the compute function will derive a new state given the `rvalue` or `operand`. For example, to perform taint analysis, we could define a tainted state for each function parameter, and propagate it to other values following assignments and expressions.

An `entrypoint` in Solana contains all the public function APIs or function calls. We traverse every function API with the following state transition. We describe the notations in Tab. 3.

- **Function Calls.** When entering a function call $\_t = F(a_1, a_2, ..., a_n)$ for function $F(x_1, x_2, ..., x_n) \rightarrow r$, we handle the epilogue by mapping the actual parameters $a_1, a_2, ..., a_n$ to formal parameters $x_1, x_2, ..., x_n$, and push the current function into the call stack CS. We then create a sub-state S_1 and process each `STATEMENT` or `TERMINATOR` in VRustMIR. Formally, we have `function prologue`:

$$\text{Prologue}\ \frac{S_0; \_t = \mathbf{F}(a_1, a_2, \cdots, a_n); \quad CS; \mathbf{F}(x_1, x_2, \cdots, x_n) \rightarrow r;}{S_1 \Rightarrow S_0 \cup \{a_i \rightarrow x_i | 1 \le i \le n\}; CS \Rightarrow CS \cup \{\mathbf{F}\}}$$

When exiting from a function, the epilogue is processed by merging the return states S_n, match the formal return value r to the actual return value $\_x$, and poping the call stack CS. Formally for `function epilogue`:

$$\text{Epilogue}\ \frac{S_n; \_t = \mathbf{F}(a_1, a_2, \cdots, a_n); \quad CS; \mathbf{F}(x_1, x_2, \cdots, x_n) \rightarrow r;}{S_{n+1} \Rightarrow S_n \cup \{r \rightarrow \_t\}; CS \Rightarrow CS - \{\mathbf{F}\}}$$

- **Basic Blocks.** We use value flow analysis to handle block closure. We process each `STATEMENT` or `TERMINATOR` inside the function body in reverse postorder, and build and update the state machine for each vulnerability pattern.

3.2 Vulnerability Patterns

VRust is capable of detecting eight kinds of vulnerabilities by enforcing the verification to specific fields (owner, signer, keys) in the AccountInfo structure (Fig. 2), as well as by examining through the Rust MIR to detect potential arithmetic overflow/underflows, account type confusion, CPI, precision errors, and bump seed canonicalization vulnerabilities. These vulnerabilities cannot be fixed in the compiler scope, instead, the smart contracts are required to be revised by adding proper checks or protections. The Rust compiler would not have knowledge of a missing check, and it would not protect the unchecked arithmetic operators in the release build.

Table 3: Notations for Vulnerability Patterns.

Notation	Description
P	Set of variables (accounts) with states.
S_i	The i-th set of states.
$S_i \vdash p$	Get the state in S_i related to an account p.
$true\langle p.x\rangle$	set field x in state of an account p into true.
$false\langle p.x\rangle$	set field x in state of an account p into false.
$\forall p \in P$	Iterate all accounts p in P.

3.2.1 *Missing Signer Check.*

This vulnerability is caused by the failure to add a check on whether a user has signed a sensitive instruction (e.g., withdrawal of tokens, or other operations that involve monetary operations) to be executed on a smart contract. Each account from the AccountInfo struct has an `is_signer` field as a flag to indicate whether an instruction requires a transaction signature matching the Pubkey, as shown in Fig. 2. Without validating the signer field, a user can fake the authority account to execute an unauthorized, unsigned transaction to withdraw funds without the consent of the authority user (Lst. 3).

```rust
fn withdraw(program_id: &Pubkey, accounts: &[AccountInfo],
↪  amount: u64)->ProgramResult {
    let acc_iter = &mut accounts.iter();
    let wallet_info = next_account_info(acc_iter)?;
    let authority_info = next_account_info(acc_iter)?;
    let destination_info = next_account_info(acc_iter)?;
    let wallet = Wallet::deserialize(&mut
↪  &(*wallet_info.data).borrow_mut()[..])?;
    assert_eq!(wallet_info.owner, program_id);
    assert_eq!(wallet.authority, *authority_info.key);
    transfer(wallet_info, destination_info, amount);
    ...
}
```

Listing 3: Example of a Missing Signer Check.

This bug can be fixed by checking whether the authority_info account requires a signature for a sensitive instruction in withdraw (Line 3 in Lst. 4). It can be accomplished by adding an assertion statement before withdrawing any funds and checking the condition of whether authority_info is an account that requires a signature in the is_signer field.

We dedicate two fields v and c in our state for signer checker, where v stands for whether a variable is a signer, and c implies if the signer variable has been checked. We report a missing signer check if a function defining an account needs a signer check $\{v : true\}$ when there is no signer check in the whole program scope $\{c : false\}$. Since the value of the is_signer field cannot be obtained with static analysis, we adopt name-based semantics to determine if an instruction demands a signer check. One challenge is to determine which account needs verification against the signer.

Our heuristic to use variable names is one simple but effective solution. It is based on observations of real-world projects, as most Solana smart contracts are developed in good programming practices and followed standard naming conventions. The evaluation results demonstrate the effectiveness of our heuristics. Specifically, we collect a list of common account names that requires a signer check as follows: `authority`, `authority_info`, `owner`, `admin`, `manager`, `admin_acc`, `owner_acc`. Our list is growing, and we will adopt more generalized naming semantics in future work. To achieve soundness and have no false negatives, human efforts are required in the loop to conduct code annotation for complete signer variable capturing.

The initial state of the signer check for each function is:

$$\frac{S_i}{\forall \quad p \in P \quad S_{i+1} \Rightarrow S_i \cup \{S_i \vdash p : \{false\langle p.v\rangle, false\langle p.c\rangle\}\}}$$

After initializing the state, we conduct a whole program traversal. For each account that needs a signer check, we update the state of that function as:

$$\text{Signer Variable Captured}\ \frac{S_i}{S_{i+1} \Rightarrow S_i \cup \{S_i \vdash p : \{true\langle p.v\rangle\}\}}$$

We collect existing signer checks by analyzing the `TerminatorKind::Call.switchint` (the MIR generated for an IF statement) and `KeyStmt::AssertionStmt` (MIR for an ASSERT statement). We update the $p.c$ field of a variable when we capture a signer check.

$$\text{Signer Check Captured}\ \frac{S_i}{S_{i+1} \Rightarrow S_i \cup \{S_i \vdash p : \{true\langle p.c\rangle\}\}}$$

For Solana smart contracts built on the Anchor framework [2], we capture a signer check by verifying if there is a function call to `Signer as anchor_lang::Accounts ::try_accounts`. Anchor checks for a signer on an account of a `Signer` type automatically.

A missing signer check will be reported for each account variable p with a state $\{p.c : false; p.v : true;\}$.

```

    assert_eq!(wallet_info.owner, program_id);
    assert_eq!(wallet.authority, *authority_info.key);
    assert!(authority_info.is_signer);

```

Listing 4: Example of Adding Field Checks.

3.2.2 Missing Owner Check. The absence of a check to determine if the user and the data involved in the transaction are owned by the smart contract (owner field of AccountInfo) exposes smart contracts to this vulnerability. Each program deployed to a Solana cluster would be available to users through a program ID – an address stored as a Pubkey type used to reference the program. An account with the type AccountInfo has an owner field corresponds to a single program ID that can write to the account, as the program owns the rights to the account. Hackers may provide accounts with fake data to withdraw money from the smart contracts that do not belong to them. The account with malicious data will have a different owner. Without any checks to determine whether a supplied account is owned by the program, the program essentially deals with untrusted data that a user can spoof.

If we remove Line 7 at Lst. 3, any user can spoof any of the accounts the function iterates through. The main vulnerability is found in wallet_info, where a user can create a fake account that points to the program's vault itself, essentially having the program withdraw from its own funds and deposit the amount to the user.

Detecting a missing owner check is similar to verifying existence of signer checks. We perform a conservative analysis that assumes every public function API with external parameters needs an owner check. We report an error if the owner field for a wallet_info account is not verified before withdrawing any funds throughout the whole program traversal.

```
pub fn verify_signatures(...) -> Result<()> {
    if *accs.instruction_acc.key !=
↪   solana_program::sysvar::instructions::id() {
        return Err(SolitaireError::InvalidSysvar
        (*accs.instruction_acc.key));
    }
    ...
```

Listing 5: Fix by Explicitly Checking Account Key.

3.2.3 Missing Key Check. Missing key check (Lst. 1) has caused a loss of $320 million from Wormhole blockchain [26, 27]. The root cause is a flaw in the Wormhole bridge's verify signatures function. It used an deprecated Solana API to parse the input account without validating the account key. This vulnerability can be fixed by either adding a key check to the account [26] (Lst. 5), or using an updated Solana API solana_program::sysvar::instructions::load_current_index_checked (Lst. 6) [27].

We detect a missing key check vulnerability by collecting all the statements accessing the data field directly, or through a `borrow_mut` method call. The account key must be properly verified before the instruction executions. In our state machine, we incorporate four fields into our state: k stands for whether a variable is a Pubkey. e indicates a variable or account is external, i.e., supplied by function argument. a implies if the data field of an account is accessed, and c signifies whether the Pubkey has been checked.

```
let current_instruction =
↪   solana_program::sysvar::instructions::
    load_current_index_checked(
    &accs.instruction_acc,
    )?;

```

Listing 6: Fix by Adopting an Updated Solana API.

The initial state for a key check is:

$$\text{Key Check Initial State}\ \frac{S_i}{\forall \quad p \in P \quad S_{i+1} \Rightarrow S_i \cup \{\mathcal{K}\}},\ \text{where}$$

$\mathcal{K} = S_i \vdash p : \{false\langle p.k\rangle, false\langle p.e\rangle, false\langle p.a\rangle, false\langle p.c\rangle\}$.

After a whole program traversal, we report errors for functions containing an external account with the type *Pubkey*, and the data field of the account is accessed without the key being verified. In our state machine, we report a missing key check if the state equals to:

$$\{p.k : true; p.e : true; p.a : true; p.c : false\}.$$

3.2.4 Integer Overflow/Underflow. Overflow vulnerabilities are unexpectedly common in Solana smart contracts as they perform math over financial data. Due to the wrapping of numeric values in Rust with 2's complement [52], Rust can still have integer overflows/underflows even though it is memory-safe. Using unchecked arithmetic operations will potentially lead to overflows and underflows, as shown in Lst. 7.

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

Listing 7: Example of Integer Overflow and Underflow.

To avoid integer underflows and overflows, Rust provides checked arithmetic operations that return a `None` type when an overflow/underflow occurs. Lst. 8 is an example of patching the previously vulnerable code.

```
**wallet_info.lamports.borrow_mut()
      .checked_sub(amount);
**destination_info.lamports.borrow_mut()
      .checked_add(amount);
```

Listing 8: Fixing Integer Overflow and Underflow.

To identify potential issues of integer overflow and underflow, we defined a state machine to contain only a single state called `external`. This state stores a Boolean value on whether the flow of external data is used in an unchecked arithmetic operation, signifying that a user can potentially exploit the operation. An issue is only reported when an arithmetic operation is discovered to have a numeric operand that has the `external` state set to true.

3.2.5 *Cross-Program Invocation (CPI).*

This vulnerability is caused by missing a validation check on a program ID. Solana allows programs to be invoked by other programs, including an account whose key equals the unique id of the invoked program. However, if the original program has no checks in validating the program being invoked, as shown in Lst. 9, a malicious program could be executed unintentionally. A way to avoid CPI vulnerabilities is to add corresponding program ID checks (Lst. 10).

The SPL Token Program v3.1.1 includes the invoked program ID checks in the library source code, making it free from the CPI vulnerability if a smart contract is built upon a version newer than v3.1.1. However, we still see real-world projects using the deprecated SPL library without explicit checks on the program ID.

```
1 let spl_token = next_account_info(account_info_iter)?;
2 invoke_signed(
3       &spl_token::instruction::transfer_checked(
4           &spl_token.key,
5           &wallet_info.key,
6           ...
```

Listing 9: Example of Cross-Program Invocation.

```
let spl_token = next_account_info(account_info_iter)?;
assert_eq!(spl_token.key, spl_token::id());
...
```

Listing 10: Example of Fixing Cross-Program Invocation.

To detect a CPI vulnerability on projects using SPL older than v3.1.1, we define our states containing two states, k indicating whether the variable is a Pubkey of a program and c indicating whether this key has been checked.

$$\frac{S_i}{\forall \quad p \in P \quad S_{i+1} \Rightarrow S_i \cup \{S_i \vdash p : \{false\langle p.k\rangle, false\langle p.c\rangle\}}$$

When coming across a variable p with PubKey type, and an SPL instruction is invoked on that variable (Line 3 in Lst. 9), we indicate that the PubKey needs to be checked by transiting the state $\{true\langle p.k\rangle\}$. Once observed an assertion with one operand equal to an SPL program id `spl_token::id()`, we update the other operand in the assertion with a state checked: $\{true\langle p.c\rangle\}$.

When the whole program traversal is finished, we examine all program ID arguments in SPL instructions and report those without a proper check:

$$\{p.k : true; p.c : false\}$$

3.2.6 *Numerical Precision Error.*

This vulnerability is caused by the round operation in transactions, letting attackers steal one smallest unit of the token at a time, causing a severe loss of funds with multiple attempts.

One situation where the numerical precision error occurs is in the spl_token lending program. In that contract, users deposit their tokens for collateral tokens and redeem them back at a certain exchange ratio. However, as the precision of the variable amount is limited (nine decimals for Solana), these deposit and redeem functions will round up to the nearest integer. As these Solana applications may also operate BTC or ETH, those precision errors could lead to severe consequences.

One way to fix this vulnerability is using floor() functions instead of round(), preventing the program from returning an amount more than the amount put in. To detect such errors, we use a simple but effective strategy of traversing the whole program and reporting an issue when variables are passed into a round() function. As most Solana smart contracts use integers instead of floating-point numbers, we observe a small number of cases involving a function call to the round() function, making it easier to examine the report manually.

3.2.7 *Bump Seed Canonicalization.*

This vulnerability is caused by failing to check the bump seed validity of program addresses, and may produce fake program addresses that result in tampering with other programs' data.

Bump seeds are 8-bit unsigned integers used as additional seeds to derive a valid PDA. PDAs are generated from a combination of seeds and a program id. If a generated address lies on the *ed25519 elliptic curve* [8], then it is a valid public key thus not a valid PDA. Solana adds an offset to the address called bump seeds to "bump" the address off the elliptic curve, and obtain a legitimate PDA. PDAs of a Solana program can have multiple valid bumps with the same seeds, giving multiple different addresses. Therefore, PDAs can be faked, and a smart contract should validate the bump seeds using `find_program_address` before executing critical transactions.

Our state machine uses naming heuristics to check bump seed issues, reporting all functions that include a variable named "bump" that can be externally modified.

3.2.8 Type Confusion. Account data in Solana are stored as byte arrays. In smart contracts, serialization and deserialization are heavily used to cast the data from and to the actual types. However, this process may deserialize data from type *A* into type *B*, as long as their memory layout is compatible. Type confusion vulnerabilities allow hackers to misinterpret data in ways they want, leading to unpredictable results such as malicious transactions and a loss of funds.

Lst. 11 shows an example. The fields of *TipPool* are of types: *Pubkey*, *u64* and *Pubkey*, occupying 32, 8 and 32 bytes separately after serialization. The first three fields of *Vault* are of types: *PubKey*, *f64* and *Pubkey*, which also occupy 32, 8 and 32 bytes, meaning that *TipPool* and *Vault* are compatible. Therefore, in function *withdraw*, *pool_info.data* could have actual type *Vault* deserialized to type *TipPool*. Since users have control of *Vault*, it is possible to bypass all checks that are supposed to restrict the *TipPool* in withdraw function, resulting in a loss of funds.

```
pub struct TipPool {
    pub withdraw_authority: Pubkey,
    pub value: u64,
    pub vault: Pubkey,
}
pub struct Vault {
    pub creator: Pubkey,
    pub fee: f64,
    pub fee_recipient: Pubkey,
    pub seed: u8,
}
fn withdraw(program_id: &Pubkey,
    accounts: &[AccountInfo],
    amount: u64) -> ProgramResult {
    let pool_info =
        next_account_info(account_info_iter)?;
    let mut pool = TipPool::deserialize(
        &mut &(*pool_info.data)
        .borrow_mut()[..])?;
    ...
}
```

Listing 11: Example of Type Confusion.

Checking this type of vulnerability is different from others. In addition to MIR, we also need to analyze the HIR [43] to find all memory layouts that are compatible with user-defined types.

(1) Collect all struct definitions in the target crate and compute all pairs of struct definitions that are compatible in memory layout.
(2) For each pair of such struct types, check if at least one of them appears in a deserialization statement in the MIR.
(3) For a deserialization statement, check if there are any type checks on the deserialized variable.

Step 3 is highly related to semantics. Currently, we rely on a few heuristics to reduce false positives. First, if there are checks on fields such as *type* and *atype* (data field in Solana framework, such as wormhole, to store custom data type information), we treat them as type checks. Second, if a slice of the deserialized data is checked against a constant array, we treat them as type checks (this is how Anchor [2] handles this issue).

4 IMPLEMENTATION

We have implemented a prototype tool, VRust, for all checkers presented in Section 3. The implementation efforts of the framework are about 6K LoC in scale on top of the Rust Compiler (rustc). To reduce the users' efforts, we also developed a script to automatically scan all crates in a repository and run the tool on only Solana smart contracts (by checking if a crate has a dependency such as *solana-program* and supporting the *bpf* architecture).

Anchor Support. Anchor is a utility library to ease the task of writing Solana smart contracts [2]. It provides an extended *Account* type that supports customized security validation (e.g., ownership check, signer check, etc.). By using derived macros defined in it, lots of boilerplate code is eliminated. Users only need to write functions for handling different logic of the smart contracts, and Anchor will automatically generate the instructions and the dispatch function. By analyzing trait function implementations such as *try_accounts* and *try_from*, VRust is able to capture all constraints defined by users.

Conditional Compiling. Since Solana smart contracts are compiled to e-BPF bytecode, some crates may have conditionals in compiling code, for example:

```
#![cfg(all(target_arch = "bpf",
          not(feature = "no-entrypoint")))]
```

This will prevent part or all smart contract code from being compiled. Our tool will automatically detect such conditionals and disable them.

Scan reports. For each crate, VRust generates a PDF report, in which all vulnerabilities and their types are listed. For each vulnerability, the report contains the vulnerable lines of code, the function, and the static call stack that reaches the code. It will also contain the vulnerability description and viable alleviation if possible (e.g., for integer overflow). This detailed report is user-friendly to verify if a reported vulnerability is potentially true.

PoC verification. For reported vulnerabilities that we are uncertain of, we developed a PoC to verify it by using an open-source Solana PoC framework [39].

5 EVALUATION AND ANALYSIS

We evaluated our VRust framework on over one hundred Solana smart contracts, and provided detailed results for 12 popular real-world projects[2] with 146,861 LoC. We collect the statistics on 12 projects for the number of vulnerabilities reported for each checker in Tab. 4.

We address the following research questions:

- RQ1: Can VRust detect existing vulnerabilities?

[2]Cashio, Quarry, Metaplex, Gem Farm, Stableswap, Wormhole, Solana Program Library (SPL), Solana-Escrow, Mango-v3, SolPayments, Neodyme Breakpoint Workshop, and Project Serum Sealevel Attacks.

- RQ2: Can VRust discover new vulnerabilities in popular Solana smart contracts?
- RQ3: What are the soundness and completeness of VRust?
- RQ4: What are the main reasons for false positives?

Table 4: Vulnerabilities Detected by VRust Checkers.

Checker	True Positives (Total Reported)
Missing Owner Check	2 (2)
Missing Signer Check	2 (2)
Missing Key Check	1 (108)
Integer Overflow/Underflow	12 (74)
Account Confusion	2 (45)
Cross Program Invocation	4 (7)
Numerical Precision Error	1 (1)
Bump Seed	1 (1)
Total	25 (240)

5.1 RQ1: Known Vulnerabilities

To begin with the experiments, we tested our framework on the Neodyme Breakpoint Workshop [38] and Project Serum Sealevel Attacks [4]. Both projects are collections of vulnerabilities prevalent in real-world projects.

The Neodyme Breakpoint Workshop introduces five common security issues in Solana. It consists of five crates with one of each vulnerability type, including a missing owner check, missing signer check, integer overflow/underflow, account confusion, and cross-program invocation bug. VRust is able to detect all of these vulnerabilities.

Project Serum Sealevel Attacks introduces ten different kinds of Solana vulnerabilities in Anchor framework. We tested our tool on five of its sub-crates whose type is within the eight patterns we found and successfully reported one missing owner check, one missing signer check, one account confusion, one cross-program invocation, and one bump seed canonicalization vulnerability.

Another motivation of VRust was through the Wormhole network and their security issues that were exploited with a missing key check. After the development of VRust, we were able to use VRust on projects known to have security issues, including the smart contract that powers the Wormhole network, to identify the tool's efficacy and effectiveness.

Initial tests have shown that VRust was capable of detecting the bug on a missing key check [26, 27]. A utility instruction in validating whether the guardian nodes have verified signatures was breached due to the Wormhole network utilizing a deprecated set of SPL functions with no checked variants, as seen in Lst. 1. By not using the recommended `load_instruction_at_checked` function, there were no checks to verify whether the user-provided accounts were SPL special sysvar instructions.

In terms of the vulnerability detection, VRust identified that the `verify_signatures` function tried to access data from a sensitive `instruction_acc` that would be executed. The account was identified as a `solana_program ::account_info::AccountInfo` type, but the data flow did not contain any validation against the keys to the account, which was flagged in the generated report for Wormhole. In addition to this true positive, VRust identified six false positives within the smart contract: one overflow from the contract and four overflows, and one missing key check from a dependency in SPL. The false positives were found to be insignificant or not exploitable.

VRust can reproduce and report a true positive in a missing key check for the Wormhole network that is available in their GitHub repository. However, some vulnerabilities involve complex semantics beyond the capability of the current VRust checkers. One example is the "Infinite Mint Glitch" discovered in Cashio [34]. Specifically, attackers can fake one crucial account used to invoke instructions, which is fixed with a key check. Cashio employs Saber [50] and Arrows [41] as collateral, which make vulnerability detection more complicated as the data field of the faked account is not accessed directly. Instead, it only uses fields like *self.common.crate_collateral_tokens* or *self.common.crate_mint*, and these fields are used by invoking new instructions, so the tool has no knowledge of the missing check. We will leave this as future work.

5.2 RQ2: New Vulnerabilities

We collected a set of popular Solana projects from Github, ran VRust on them, and manually verified the reports. The results are shown in Table 5. We found 12 potential vulnerabilities and have reported them to project developers. We discuss them separately below. For security concerns, we only release the vulnerability details after they have been confirmed and fixed by the developers.

Quarry. Quarry is an open protocol for launching liquidity mining programs on Solana. There are 6 crates in this project, and 1 potential integer overflow is reported in *quarry_registry.*

SPL. Solana Program Library (SPL) is a collection of Solana-maintained on-chain programs. There are 27 crates analyzed in total, among them, the tool reported five potential vulnerabilities: three integer overflows in *binary-option*, one integer overflow in *spl-record*, one integer overflow in *spl-stake-pool* and one wrong key check in *spl-governance.*

The 3 integer overflows in *binary-option* were reported to the SPL developers, confirming they are notable vulnerabilities. As noted by one of the developers: "*Thanks for pointing out all of these issues. They are certainly critical and should be fixed at the program level. Thankfully, binary-option is a reference that we do not publish to mainnet. If your students can put in pull requests to resolve these issues, I'll be very happy to merge them.*"

One of the overflow issues allows the user to manipulate the prices of buying and selling a transaction on binary options under certain conditions through user-inputted accounts. The vulnerable function is said to be called by other higher-level protocols, as mentioned in the documentation. The other issues come from an overflow on a set of mathematical operations that result in receiving less and more of an expected stake pool amount.

Metaplex. Metaplex is a protocol and application framework for decentralized NFT minting, storefronts, and sales. There are 9 crates analyzed in this project and the tool reported 3 potential vulnerabilities: 1 integer overflow in *token-metadata*, 1 integer

Table 5: Evaluation Results of Solana Projects on GitHub.

Project	Description	LoC	Potential Vulnerabilities (Total Reported)
Quarry	An open protocol for launching liquidity mining programs on Solana.	4,303	1 (1)
SPL	Solana program library.	85,591	5 (60)
Metaplex	Protocol and application framework for decentralized NFT minting, storefronts, and sales.	8,967	3 (41)
Gem Farm	Configurable staking for NFT Projects on Solana.	3,909	1 (16)
StableSwap	an automated market maker for mean-reverting trading pairs.	8,256	0 (7)
Mango v3	Mango V3 provides support for decentralized margin trading.	18,526	0 (21)
Solana-Escrow	An escrow smart contract built for the Solana blockchain.	320	0 (2)
SolPayments	Protocol to receive cryptocurrency payments for online shops.	3,331	2 (12)
Total		133,203	12(160)

overflow in *mpl-gumdrop* and 1 integer overflow in *mpl-candy-machine.*

Gem Farm. Gem Farm is a configurable staking contract for NFT Projects on Solana. There are two crates analyzed in this project. VRust reported 1 potential integer overflow vulnerability (in *gem_farm*).

StableSwap. StableSwap is a Solana token swap program. There are five crates analyzed, and there is no potential vulnerability found after our inspection.

SolPayments. SolPayments is a protocol that allows online shops to receive cryptocurrency payments. There is only one crate, and two cross-program invocations were reported as critical vulnerabilities.

Mango v3. Through a robust risk engine, Mango provides a single platform for lending, borrowing, swapping, and leveraging trading crypto assets for the Solana blockchain. VRust reports 21 overflows that are considered false positives.

Solana-Escrow An escrow program written in Rust for asset exchange on Solana. VRust reports two missing key checks for this repository that are considered false positives.

5.3 RQ3: Soundness and Completeness

VRust does not offer a guarantee of soundness or completeness. We analyzed the false positives and organized the false positive patterns in Section 5.4. As a research prototype, VRust could miss real vulnerabilities if none of the vulnerability patterns are matched. In this case, adding a new pattern to VRust is easy. All we have to do is design new states and organize the state transits.

Although we do not guarantee soundness and completeness, VRust performs a very conservative value flow analysis that puts more weight on false negatives. We build VRust to analyze the Rust MIR, which abstracts the patterns more broadly. User-defined functions and Solana programming libraries will have common patterns at the MIR level, which enables VRust to be capable of capturing more potential vulnerabilities.

5.4 RQ4: False Positive Patterns

We organize common false positive patterns reported by VRust. By incorporating those patterns into the VRust architecture, we can eliminate the majority of false positives. For example, we are able to bring down the number of false positives from 17 to 5 in Wormhole project.

5.4.1 *Intended integer overflow.*

On StableSwap[3], a few integer overflows with the same root cause are reported:

```
pub fn overflowing_mul(self, other: $name) -> ($name,
↪  bool) {
$crate::uint_overflowing_mul!($name, $n_words, self,
↪  other)
}

```

The `uint_overflowing_mul` will eventually call `split_u128`:

```
let (hi, low) = Self::split_u128(a as u128 * b as u128);
```

Variable a and b are of type u64. After converting to u128 type, an overflow would not happen. The crate will also use the higher bit to indicate if there could be an overflow on the original type, which will set the bool result of `overflowing_mul`.

5.4.2 *False integer overflow due to implicit constraints on the operands.*

There are a few cases where the reported integer overflows are false positives because operands has constraints.

(1) Use checked_rem[4]

```
      Ok(7 - offset.checked_rem(8)
      .ok_or(MetadataError:: Overflow)? as u32)
```

Due to the checked_rem(8), the right-hand side will always be less or equal to 7, and thus not an overflow.

(2) Type upscale[5]

```
let token_type_count =
↪  Vault::get_token_type_count(vault_info)
    .checked_div(8)
    .ok_or(NumericalOverflowError)?;
if bid_redemption_info.data_is_empty() {
```

[3]https://github.com/saber-hq/stable-swap
[4]state.rs in mpl-token-metadata (metaplex)
[5]utils.rs in mpl-metaplex (metaplex)

```
    create_or_allocate_account_raw(
        1 + 9 + 32 + 1 + token_type_count as usize,
```

The variable token_type_count is of type u8, and therefore, it cannot cause overflow with type upscale.

(3) AccountInfo Heuristics[6]

```
pub fn from_account_info(a: &AccountInfo,
    ) -> Result<TokenTypeTracker, ProgramError> {
let amount_type =
↪  TokenTypeTracker::get_amount_type(a)?;
offset += amount_type as usize;
```

Since a is of AccountInfo type, the data length is limited and constrained by the AccountInfo structure. The arithmetic calculation `offset += amount_type as usize;` will not cause overflow.

5.4.3 False Alarm on Key Check, caused by access to data. There are function calls that try to access the length of the data instead of the data field of an account. As a conservative detector, VRust will report a potential missing key check if the data length is accessed.

```
pub fn data_is_empty(&self) -> bool {
  self.data.borrow().is_empty()
}
pub fn data_len(&self) -> usize {
 self.data.borrow().len()
}
```

For Solana-escrow, the variable of which the data field is accessed contains multiple accounts, and we check the key of one of those accounts. We still report a false positive for this variable as a conservative analysis framework. This is also a potential source of false positives.

5.4.4 Function Initialization. VRust will report a missing key check if there is a function call to `invoke_signed` and access the data field. It would be a false positive if the function is responsible for initializing a new contract on the blockchain with no influence on previous contracts, and no transaction is involved. Therefore, hackers cannot take advantage of it.

5.4.5 Ownership Checks. There are missing ownership checks flagged incorrectly for the Solana Token Program and the Solana Token 2022 Program due to a special instruction for this SPL program. Besides the initialization of various accounts, all instructions call a validate_owner function before invoking sensitive methods. An example of `validate_owner` is from the processor.rs in SPL[7].

6 LIMITATIONS

Alias analysis. Currently, we only have intra-procedural alias analysis that is supported by the local data-flow analysis. This is used to determine if a value in MIR points to a field of interest and to resolve call targets. We have not observed a case caused by an indirect function call or inter-procedural alias relation. Moreover, for alias relation, since we have state propagation for call parameters and returns, it supports all the checkers we developed. Global variables could cause problems but are rare in Solana smart contracts.

Dependent crates. A Solana smart contract has tens or hundreds of dependent crates. By default, our analysis will enter every function reachable from the entrypoint. However, this is slow as Rust's *std* library is huge. Our solution was to mark functions in popular and safe crates such as the *std* library, *curve25519_dalek* and *bytes*, to be ignored in a blacklist.

Heuristic patterns. Vulnerabilities in smart contracts are mostly logic bugs. Some vulnerabilities are eliminated by complex solutions without compromising safety. However, our tool could still report it as a vulnerability. Nevertheless, this enables VRust to discover both known and unknown vulnerabilities in Solana smart contracts, with a reasonable number of false positives. The comprehensive PDF report generated by VRust allows the developer to easily verify them.

PoC generation. For every potential vulnerability that we reported to developers, we manually wrote a PoC to verify them. One task for future work is to automate this process: given the call stack and the vulnerable code location, the PoC can be automatically generated by extracting the smart contract instruction that reaches the vulnerable location and all the accounts that are required. Symbolic execution may be necessary to bypass branch conditions.

7 RELATED WORK

To the best of our knowledge, this work is the first end-to-end vulnerability detector targeting the Solana blockchain. We provide an overview of previous work on Solana, Ethereum, and Hyperledger Fabric.

7.1 Solana Smart Contracts

Bodziony et al. [9] proposed an aliasing system on Solana that allows users to create aliases for accounts and tokens in place of opaque addresses. Li et al. [32] presented a survey on the transition from Bitcoin to Solana for enterprise applications targeting scalability and performance. Duffy et al. [15] investigated the high TPS rate on Solana to support large-scale business applications that characterize IoT applications.

7.2 Ethereum Smart Contracts

Existing research projects focus on static analysis [19, 22, 24, 53, 61], dynamic analysis [33], symbolic execution [14, 31, 35, 37, 54], and a deep learning approach [20, 21, 42] to detect vulnerabilities for Ethereum smart contracts.

• **Static Analysis.** Slither [19] is a framework that translates Solidity smart contracts into SlithIR while keeping semantic information. Ghaleb and Pattabiraman [22] proposed SolidiFI, a tool that evaluates smart contract bug detection tools. Tikhomirov et al. [53] presented SmartCheck, an extensible tool that detects vulnerabilities in a comprehensive classification of Solidity code defects. Hajdu and Jovanović [24] proposed solc-verify, a source-level verification tool that uses modular program analysis and SMT solvers to discharge verification criteria. Zhang et al. [61] proposed SMARTSHIELD, a bytecode rectification system, to automatically

[6]state.rs in mpl-metaplex (metaplex)
[7]https://github.com/solana-labs/solana-program-library/blob/e59a0dc18626de2fca2a2acebffe3fa7b4819171/token/program/src/processor.rs#L975

fix three common security-related bugs and assist developers in releasing secure contracts.

• **Dynamic Analysis.** Liu et al. [33] focused on the reentrancy bug that was responsible for the DAO attack resulting in a $60 million loss. They proposed ReGuard, a fuzzing-based analyzer that detects reentrancy vulnerabilities.

• **Symbolic Execution.** Luu et al. [35] built a tool, Oyente, to find potential security vulnerabilities. Securify [54] is a scalable and fully automated security analyzer that can verify whether contract behaviors are safe or dangerous depending on particular attributes. ZEUS [31] is a framework for validating the validity and fairness of smart contracts. It adopts abstract interpretation, symbolic model checking, and constrained horn clauses. Mythril [14] is a security analysis framework for EVM bytecode using symbolic execution, SMT solving, and taint analysis to detect a variety of security vulnerabilities. Manticore [37] is a dynamic symbolic execution framework for analyzing binaries and Ethereum.

• **Deep Learning.** Gao et al. [20, 21] proposed SMARTEMBED, a tool that detects clone-related problems by comparing the similarities between the code embedding vectors. Qian et al. [42] detected reentrancy problems using a bidirectional long-short term memory network.

7.3 Hyperledger Fabric Smart Contracts

Two major approaches involved to analyze Fabric smart contracts are static analysis [36, 45, 59] and formal verification[7, 23].

• **Static Analysis.** Yamashita et al. [59] designed a Chaincode Analyzer based on the Abstract Syntax Tree and identified 14 risks related to non-determinism and logical risk. Based on abstract syntax tree, package dependency, and functional dependency, Lv et al. [36] implemented a detection system that can detect 16 kinds of potential vulnerabilities on Fabric. ReviveCC [45] enables the security analysis for any chaincode file based on Go's static analysis tool Revive.

• **Formal Verification.** Beckert et al. [7] presented an approach for deductive verification based on semi-interactive theorem prover KeY. Graf et al. [23] focuses on formalizing the accountability in context of Fabric smart contracts.

8 CONCLUSION

In this paper, we are the first to design and implement an end-to-end vulnerability detector for Solana smart contracts. VRust can detect eight bug patterns and produce detailed vulnerability audition reports without any code annotations. We design a novel state machine to analyze the source code that is translated into Rust MIR with inference rules. Our tool has revealed 12 previously unknown vulnerabilities and has revealed three critical vulnerabilities in the official Solana Programming Library confirmed by developers.

ACKNOWLEDGMENTS

We thank the anonymous CCS reviewers for their constructive feedback on earlier versions of this paper. This work was supported by NSF award CCF-1901482.

REFERENCES

[1] Maher Alharby, Amjad Aldweesh, and Aad Van Moorsel. 2018. Blockchain-based smart contracts: A systematic mapping study of academic research (2018). In *2018 International Conference on Cloud Computing, Big Data and Blockchain (ICCBB)*. IEEE, 1–6.
[2] Anchor. 2020. *Anchor.* Retrieved April 18, 2022 from https://hackmd.io/@ironaddicteddog/solana-anchor-escrow
[3] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. 2018. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the thirteenth EuroSys conference*. 1–15.
[4] et al. Armani Ferrante. 2022. *coral-xyz/sealevel-attacks: Common Security Exploits and Protections on Solana.* Retrieved Sep. 6, 2022 from https://github.com/coral-xyz/sealevel-attacks
[5] Avyan. 2022. *Solana vs Ethereum: A Detailed Comparison | Alexandria.* Retrieved April 05, 2022 from https://coinmarketcap.com/alexandria/article/solana-vs-ethereum-a-detailed-comparison
[6] AWS. 2022. *What is Hyperledger Fabric?* Retrieved August 10, 2022 from https://aws.amazon.com/blockchain/what-is-hyperledger-fabric/
[7] Bernhard Beckert, Mihai Herda, Michael Kirsten, and Jonas Schiffl. 2018. Formal specification and verification of Hyperledger Fabric chaincode. In *3rd Symposium on Distributed Ledger Technology (SDLT-2018) co-located with ICFEM*. 44–48.
[8] Daniel J Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. 2012. High-speed high-security signatures. *Journal of cryptographic engineering* 2, 2 (2012), 77–89.
[9] Norbert Bodziony, Paweł Jemioło, Krzysztof Kluza, and Marek R Ogiela. 2021. Blockchain-based address alias system. *Journal of Theoretical and Applied Electronic Commerce Research* 16, 5 (2021), 1280–1296.
[10] Miguel Castro, Barbara Liskov, et al. 1999. Practical byzantine fault tolerance. In *OsDI*, Vol. 99. 173–186.
[11] Yan Chen and Cristiano Bellavitis. 2020. Blockchain disruption and decentralized finance: The rise of decentralized business models. *Journal of Business Venturing Insights* 13 (2020), e00151.
[12] Usman W Chohan. 2021. Non-fungible tokens: Blockchains, scarcity, and value. *Critical Blockchain Research Initiative (CBRI) Working Papers* (2021).
[13] Coinmarketcap. 2022. *Ethereum price today, ETH to USD live, marketcap and chart | CoinMarketCap.* Retrieved April 20, 2022 from https://coinmarketcap.com/currencies/ethereum/
[14] ConsenSys. 2022. *ConsenSys/mythril: Security analysis tool for EVM bytecode. Supports smart contracts built for Ethereum, Hedera, Quorum, Vechain, Roostock, Tron and other EVM-compatible blockchains.* Retrieved April 13, 2022 from https://github.com/ConsenSys/mythril
[15] Fintan Duffy, Malika Bendechache, and Irina Tal. 2021. Can Solana's high throughput be an enabler for IoT?. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, 615–621.
[16] ethereum. 2015. *Home | ethereum.org.* Retrieved March 29, 2022 from https://ethereum.org/en/
[17] Fabric. 2022. *Fabric Blockchain.* Retrieved August 9, 2022 from https://hyperledger-fabric.readthedocs.io/en/release-2.2/blockchain.html
[18] Fabric. 2022. *Fabric Model.* Retrieved August 9, 2022 from https://hyperledger-fabric.readthedocs.io/en/release-2.2/fabric_model.html
[19] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: a static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 8–15.
[20] Zhipeng Gao, Vinoj Jayasundara, Lingxiao Jiang, Xin Xia, David Lo, and John Grundy. 2019. Smartembed: A tool for clone and bug detection in smart contracts through structural code embedding. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 394–397.
[21] Zhipeng Gao, Lingxiao Jiang, Xin Xia, David Lo, and John Grundy. 2020. Checking smart contracts with structural code embedding. *IEEE Transactions on Software Engineering* (2020).
[22] Asem Ghaleb and Karthik Pattabiraman. 2020. How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 415–427.
[23] Mike Graf, Ralf Küsters, and Daniel Rausch. 2020. Accountability in a permissioned blockchain: Formal analysis of hyperledger fabric. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 236–255.
[24] Ákos Hajdu and Dejan Jovanović. 2019. solc-verify: A modular verifier for solidity smart contracts. In *Working Conference on Verified Software: Theories, Tools, and Experiments*. Springer, 161–179.
[25] Jim Hendler. 2009. Web 3.0 Emerging. *Computer* 42, 1 (2009), 111–113.
[26] Hendrik Hofstadt. 2022. *Check instructions sysvar · certusone/wormhole@e8b9181.* Retrieved April 26, 2022 from https://github.com/certusone/wormhole/commit/e8b91810a9bb35c3c139f86b4d0795432d647305

[27] Hendrik Hofstadt. 2022. *Update Solana to 1.9.4*. Retrieved March 29, 2022 from https://github.com/certusone/wormhole/commit/7edbbd3677ee6ca681be8722a607bc576a3912c8

[28] Hyperledger. 2022. *Hyperledger Architecture, Volume II*. Retrieved August 10, 2022 from https://www.hyperledger.org/wp-content/uploads/2018/04/Hyperledger_Arch_WG_Paper_2_SmartContracts.pdf

[29] Hyperledger. 2022. *Hyperledger – Open Source Blockchain Technologies*. Retrieved August 9, 2022 from https://www.hyperledger.org/

[30] Connor Dempsey Justin Mart. 2021. *Scaling Ethereum & crypto for a billion users | by Coinbase | The Coinbase Blog*. Retrieved March 29, 2022 from https://blog.coinbase.com/scaling-ethereum-crypto-for-a-billion-users-715ce15afc0b

[31] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. Zeus: analyzing safety of smart contracts.. In *Ndss*. 1–12.

[32] Xiangyu Li, Xinyu Wang, Tingli Kong, Junhao Zheng, and Min Luo. 2021. From Bitcoin to Solana–Innovating Blockchain Towards Enterprise Applications. In *International Conference on Blockchain*. Springer, 74–100.

[33] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. 2018. Reguard: finding reentrancy bugs in smart contracts. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. IEEE, 65–68.

[34] lunaray. 2022. *A hacker exploited an "infinite mint glitch" and drained about $28 million worth of assets from… | by lunaray | Coinmonks | Mar, 2022 | Medium*. Retrieved April 30, 2022 from https://medium.com/coinmonks/a-hacker-exploited-an-infinite-mint-glitch-and-drained-about-28-million-worth-of-assets-from-a19277c0e20c

[35] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 254–269.

[36] Penghui Lv, Yu Wang, YaZhe Wang, and Qihui Zhou. 2021. Potential Risk Detection System of Hyperledger Fabric Smart Contract based on Static Analysis. In *2021 IEEE Symposium on Computers and Communications (ISCC)*. IEEE, 1–7.

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

[38] Neodyme. 2021. *Introduction - Solana Security Workshop*. Retrieved Sep 6, 2022 from https://workshop.neodyme.io/

[39] neodyme labs. 2021. *solana-poc-framework*. Retrieved April 26, 2022 from https://github.com/neodyme-labs/solana-poc-framework

[40] Poly Network. 2022. *Rekt - Poly Network - REKT*. Retrieved August 10, 2022 from https://rekt.news/polynetwork-rekt/

[41] Arrow Protocol. 2021. *Arrow*. Retrieved April 30, 2022 from https://arrowprotocol.com/

[42] Peng Qian, Zhenguang Liu, Qinming He, Roger Zimmermann, and Xun Wang. 2020. Towards automated reentrancy detection for smart contracts based on sequential models. *IEEE Access* 8 (2020), 19685–19695.

[43] RFCs. 2015. *1191-hir - The Rust RFC Book*. Retrieved April 27, 2022 from https://rust-lang.github.io/rfcs/1191-hir.html

[44] RFCs. 2022. *1211-mir - The Rust RFC Book*. Retrieved April 21, 2022 from https://rust-lang.github.io/rfcs/1211-mir.html

[45] sivachokkapu. 2020. *ReviveCC*. Retrieved August 11, 2022 from https://github.com/sivachokkapu/revive-cc

[46] Solana. 2019. *Scalable Blockchain Infrastructure: Billions of transactions & counting | Solana: Build crypto apps that scale*. Retrieved March 29, 2022 from https://solana.com/

[47] Solana. 2022. *Calling Between Programs | Solana Docs*. Retrieved July 31, 2022 from https://docs.solana.com/developing/programming-model/calling-between-programs

[48] solana. 2022. *Overview | Solana Docs*. Retrieved April 07, 2022 from https://docs.solana.com/developing/programming-model/overview

[49] Solana. 2022. *Program Derived Addresses (PDAs) | Solana Cookbook*. Retrieved August 8, 2022 from https://solanacookbook.com/core-concepts/pdas.html#facts

[50] Solana. 2022. *Saber | Solana AMM and DEX*. Retrieved April 30, 2022 from https://saber.so/

[51] Solana. 2022. *Tower BFT | Solana Docs*. Retrieved April 05, 2022 from https://docs.solana.com/implemented-proposals/tower-bft

[52] Carol Nichols Steve Klabnik. 2022. *Data Types - The Rust Programming Language*. Retrieved April 05, 2022 from https://doc.rust-lang.org/book/ch03-02-data-types.html#integer-overflow

[53] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. Smartcheck: Static analysis of ethereum smart contracts. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*. 9–16.

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

[55] John Wang. 2021. *Parallel Processing: Solana's Key to Hardware Scalability*. Retrieved April 11, 2022 from https://www.johnwang.xyz/parallel-processing-solanas-key-to-hardware-scalability/

[56] Alex White-Gomez. 2022. *Solana vs Ethereum: What's the Difference?* Retrieved April 04, 2022 from https://www.one37pm.com/nft/tech/solana-vs-ethereum

[57] Anatoly Yakovenko. 2018. Solana: A new architecture for a high performance blockchain v0. 8.13. *Whitepaper* (2018).

[58] Anatoly Yakovenko. 2019. *Sealevel — Parallel Processing Thousands of Smart Contracts | by Anatoly Yakovenko | Solana | Medium*. Retrieved April 11, 2022 from https://medium.com/solana-labs/sealevel-parallel-processing-thousands-of-smart-contracts-d814b378192

[59] Kazuhiro Yamashita, Yoshihide Nomura, Ence Zhou, Bingfeng Pi, and Sun Jun. 2019. Potential risks of hyperledger fabric smart contracts. In *2019 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, 1–10.

[60] Ycharts. 2022. *Bitcoin Market Cap*. Retrieved April 04, 2022 from https://ycharts.com/indicators/bitcoin_market_cap

[61] Yuyao Zhang, Siqi Ma, Juanru Li, Kailai Li, Surya Nepal, and Dawu Gu. 2020. Smartshield: Automatic smart contract protection made easy. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 23–34.