

# BELOBOG: Move Language Fuzzing Framework For Real-World Smart Contracts

WANXU XIA\*, National Superior College for Engineers, Beihang University, China

ZIQIAO KONG\*, Nanyang Technological University, Singapore

ZHENGWEI LI, Bitslab, Singapore

YI LU, Movebit, Singapore

PAN LI, Bitslab, Singapore

LIQUN YANG, School of Cyber Science and Technology, Beihang University, China

YANG LIU, Nanyang Technological University, Singapore

XIAPU LUO, The Hong Kong Polytechnic University, China

SHAOHUA LI✉, The Chinese University of Hong Kong, China

Move is a research-oriented programming language design for secure and verifiable smart contract development and has been widely used in managing billions of digital assets in blockchains, such as Sui and Aptos. Move features a strong static type system and explicit resource semantics to enforce safety properties such as the prevention of data races, invalid asset transfers, and entry vulnerabilities. However, smart contracts written in Move may still contain certain vulnerabilities that are beyond the reach of its type system. It is thus essential to validate Move smart contracts. Unfortunately, due to its strong type system, existing smart contract fuzzers are ineffective in producing syntactically or semantically valid transactions to test Move smart contracts.

This paper introduces the first fuzzing framework, BELOBOG, for Move smart contracts. BELOBOG is type-aware and ensures that all generated and mutated transactions are well-typed. More specifically, for a target Move smart contract, BELOBOG first constructs a *type graph* based on Move's type system, and then generates or mutates a transaction based on the *graph trace* derived from the *type graph*. In order to overcome the complex checks in Move smart contracts, we further design and implement a concolic executor in BELOBOG.

We evaluated BELOBOG on 109 real-world Move smart contract projects. The experimental results show that BELOBOG is able to detect 100% critical and 79% major vulnerabilities manually audited by human experts. We further selected two recent notorious incidents in Move smart contracts, *i.e.*, *Cetus* and *Nemo*. BELOBOG successfully reproduced full exploits for both of them, without any prior knowledge.

CCS Concepts: • **Security and privacy** → **Software and application security**.

Additional Key Words and Phrases: Fuzzing, Smart Contract, Blockchain

## 1 Introduction

Smart contracts have become a core component of modern blockchain systems, enabling decentralized applications to automate interactions. Since smart contracts have been used in managing numerous digital assets, their correctness and security are critical. Bugs in deployed smart contracts can result in severe consequences, such as financial loss [36]. The most prevailing smart contract languages, Solidity [1], adopt an easy-to-use language design but expose developers to subtle

---

\*Both authors contributed equally to this research.

✉Corresponding author.

---

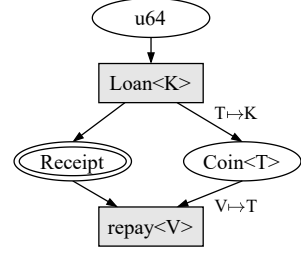
Authors' Contact Information: Wanxu Xia, ysiel@buaa.edu.cn, National Superior College for Engineers, Beihang University, Beijing, Beijing, China; Ziqiao Kong, ziqiao001@e.ntu.edu.sg, Nanyang Technological University, Singapore, Singapore, Singapore; Zhengwei Li, Bitslab, Singapore, Singapore, robin@bitslab.xyz; Yi Lu, Movebit, Singapore, Singapore, y@movebit.xyz; Pan Li, Bitslab, Singapore, Singapore, paul@bitslab.xyz; Liqun Yang, lqyang@buaa.edu.cn, School of Cyber Science and Technology, Beihang University, Beijing, Beijing, China; Yang Liu, yangliu@ntu.edu.sg, Nanyang Technological University, Singapore, Singapore, Singapore; Xiapu Luo, csxluo@comp.polyu.edu.hk, The Hong Kong Polytechnic University, Hong Kong, Hong Kong, China; Shaohua Li, shaohuali@cuhk.edu.hk, The Chinese University of Hong Kong, Hong Kong, Hong Kong, China.

```

1  module pool;
2  // This struct type has "drop" and "store" abilities.
3  public struct VeryAble has drop,store {...}
4  // This struct type has neither "drop" nor "store" abilities.
5  public struct Receipt<T> {...}
6
7  public fun loan<T>(amount: u64): (Coin<T>, Receipt<T>) {
8    // Function to flash loan.
9    ...
10 }
11 public fun repay<T>(coin: Coin<T>, receipt: Receipt<T>) {
12   // Function to repay flash loan with coins.
13   ...
14 }

```

(a) A constructed Move smart contract with two struct definitions and two function definitions.



(b) The simplified type graph for the constructed Move smart contract.

Fig. 1. A constructed Move smart contract (left) and the built type graph for it (right).

security pitfalls, such as reentrancy attacks. These recurring issues highlight the need for languages that provide stronger security guarantees by design.

The Move language emerges as a new smart contract language design with strong type safety. Its static type system and resource-oriented semantics enforce key safety properties at both compile and run time, preventing common bugs like reentrancy. More and more mainstream blockchains have chosen Move as their language for developing smart contracts, such as Sui[28] and Aptos[5]. Despite its strong type system, smart contracts written in Move can still contain logic flaws or other bugs that are beyond the scope of the type system, such as the notorious bugs in *Cetus* [4] and *Nemo* [9]. Therefore, it is still essential to test and verify a Move smart contract. Unfortunately, existing smart contract fuzzers are largely designed for the Ethereum Virtual Machine (EVM)[29] and lack awareness of Move’s unique type system. As our evaluation in Section 2.1 will show, they all fail to generate enough valid transactions — most generated transactions are rejected by the Move Virtual Machine before reaching meaningful code paths.

Generating diverse yet valid transactions is crucial for a successful fuzzer design. We identify three key challenges in fuzzing Move smart contracts, namely *type safety of objects*, *type parameter of functions*, and *type abilities of objects*. Figure 1a shows a constructed Move smart contract that exhibits all three challenges.

**Challenge 1: Type safety of objects.** Move represents on-chain states as strongly-typed objects, which results in several strict rules for objects, *i.e.*, (1) objects that are passed as function inputs must match the function’s signatures, (2) objects can only be created in the module where they were defined, and (3) a function caller can only use its owned or shared objects to interact with other functions. Thus, fuzzers cannot fill random bytes to forge objects but must instead construct them through legitimate function calls or reuse existing ones. For example, the second argument type of `repay<T>()` in Figure 1a is `Receipt<T>`, which is defined in the same module as `repay<T>()`. In this module, the only function that can produce objects of type `Receipt<T>` is `loan<T>()`, and thus a valid transaction must call `loan<T>()` first to create a usable object of type `Receipt<T>` before calling `loan<T>()`.

**Challenge 2: Type parameter of functions.** Many Move functions and structs are generic. Fuzzers must provide valid concrete type instantiations to generate complete and executable transactions, which existing fuzzers fail to do. For example, both functions `loan<T>()` and `repay<T>()` accept an extra type parameter `T`, which usually specifies the actual currency coin types on blockchains,

such as USDC and USDT. When constructing a transaction, it is crucial to find the valid concrete types for all type parameters.

**Challenge 3: Type abilities of objects.** All Move objects can have several associated abilities to constrain how they can be used and persisted. For example, the struct type `VeryAble` in Figure 1a has `drop` and `store` abilities while `Receipt` has none of them. The `drop` and `store` are two abilities that may affect the validity of a transaction. The `store` ability determines if an object can be stored in persistent storage, while the `drop` ability allows an object to be dropped without being used. An object without `drop` and `store` abilities, such as `Receipt`, can not be transferred to other accounts and thus must be consumed by *one and only one function call* within the same transaction. Such objects are usually called “**Hot Potato**” and are typically used for implementing flash loans [21]. It enables users to perform permissionless and collateral-free loans and ensures that the loans are repaid within a single transaction. When generating a transaction, we have to guarantee that all produced “Hot Potato” objects are properly handled. For example, if we create a function call `loan<T>()` in a transaction, then the created object of type `Receipt<T>` should be passed as input to another function to consume it. Such “Hot Potato” objects introduce constraints across different function calls.

**Our contributions.** In this paper, we propose a *type graph*-guided Move fuzzing framework, BELOBOG. Given a Move smart contract, we build a *type graph* to model all the type constraints. This type graph will later be used to guide the generation and mutation of transactions during fuzzing. For example, Figure 1b shows a simplified type graph for the smart contract in Figure 1a. Nodes in the graph represent different object types or functions, while the annotated edges maintain different relations and constraints among them. In order to further overcome tricky checks in Move smart contracts, we also design and implement a concolic executor in BELOBOG.

We extensively evaluate BELOBOG on a large-scale dataset provided by a well-known Web3 audit company, Movebit. and a real-world dataset collected by an empirical work [25]. The results are very promising: BELOBOG is able to detect 100% critical vulnerabilities and 79% major vulnerabilities when cross-compared with manually audited results by human experts. We also selected two notorious incidents in the Move ecosystem, *Cetus* with \$200 million loss and *Nemo* with \$3 million loss, and BELOBOG successfully reproduced both incidents with full exploits. We envision that this work bridges the gap between Move’s static safety guarantees and dynamic testing and provides a general framework for improving the overall security and reliability of Move-like smart contract ecosystems. We have open-sourced BELOBOG at <https://github.com/abortfuzz/belobog>.

## 2 Motivation

In this section, we motivate our design by first showing the deficiency of existing fuzzers on Move smart contracts, then empirically demonstrating the criticality of handling the special constraints in Move, and lastly exemplifying how BELOBOG can generate valid transactions with the guidance of type graph.

### 2.1 Deficiency of Existing Fuzzing Tools on Move

ITYFUZZ and SUI-FUZZER are two state-of-the-art smart contract fuzzers that can be used on Move smart contracts. We create a Move smart contract with the example shown in Figure 1a. This example, although constructed, includes the main and commonly seen features in Move smart contracts. We try to run both ITYFUZZ and SUI-FUZZER on this smart contract. Unfortunately, all the transactions generated by them have type-mismatch errors and are rejected without execution, such as the one shown in Figure 2a. The root cause is that neither of them supports type parameters, which are used in the structs and functions in our example.

```
v1, v2 = loan<T>(100);
```

(a) Both ITrFuzz and SUI-FUZZER cannot handle type parameter `<T>`.

```
v1, v2 = loan(100);
v3, v4 = loan(100);
repay(v1, v4);
```

(b) After removing type parameters, SUI-FUZZER still missed the “Hot Potato” `v2`.

```
v1, v2 = loan<USDC>(100);
repay<USDC>(v1, v2);
```

(c) BELOBOG successfully generate a valid transaction by correctly instantiating type parameters and handling “Hot Potato”.

Fig. 3. Fuzzer-generated transactions.

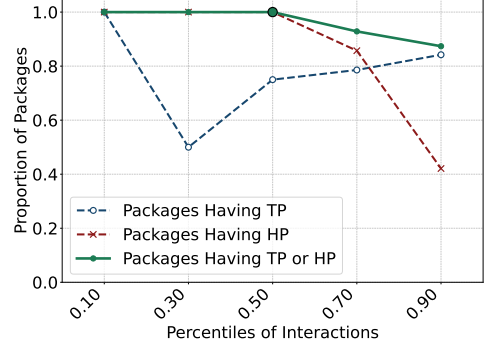


Fig. 4. Ratio of the packages having type parameters (denoted as TP) and “Hot Potato” (denoted as HP)

We further simplify the smart contract by removing all the type parameters and the type `Coin<T>` from a standard library. However, SUI-FUZZER still fails to generate any valid transactions, as shown in Figure 2b. The reason is two-fold. First, it will internally forge objects by filling random bytes, which can easily violate the type safety rule of `Move`. Second, it is not aware of the type abilities of objects and thus “Hot Potato” objects are not properly handled. ITrFuzz generates some valid transactions in this simplified smart contract by randomly mutating function calls until all “Hot Potato” objects are handled. This random mechanism is highly inefficient and can only produce “Hot Potato” objects without type parameters. Although existing fuzzers fail to generate valid transactions on smart contracts with type parameters and “Hot Potato”, we still lack understanding of how common such features are in the real world. In order to measure the prevalence of type parameters and “Hot Potato”, we collected all *two billion* transactions on the Sui blockchain from November 2024 to November 2025. Since `Move` smart contracts are organized as packages, we obtain the source packages that each transaction uses. For all the packages, we count the number of packages having type parameters and “Hot Potato”. Figure 4 shows the percentage of packages that have type parameters, “Hot Potato”, and at least one of them. Packages are ranked based on the number of `Move` function calls that use them. The result shows that among the most used (90%) packages, more than 80% of them have at least type parameter or “Hot Potato”, indicating the prevalence of such `Move` features in real-world.

## 2.2 Effectiveness of BELOBOG

BELOBOG overcomes the limitations of previous fuzzers by implementing type-aware transaction generation and mutation. With the statically built type graph for a smart contract, BELOBOG can always generate valid transactions. For example, to generate a transaction for Figure 1a, BELOBOG first queries the type graph to understand that the output object of `loan<T>()` contains a “Hot Potato”, which can be luckily consumed by `repay<K>()`. Since there are no additional type constraints, the two type parameters `T` and `K` can be instantiated with the same concrete type `USDC`. With all the information obtained from the type graph, BELOBOG can always produce valid transactions for testing this smart contract, as shown in Figure 2c.

### 3 Methodology

In this section, we first introduce some preliminary definitions to facilitate our methodology description, and then we introduce the high-level workflow of BELOBOG. We elaborate on the technical details of the core components in the subsequent parts.

#### 3.1 Preliminaries

We make the following definitions to facilitate our description of the methodology.

*Definition 3.1 (Object).* An object refers to a typed entity. Each object is an instance of a Move struct or enum type that possesses the key abilities.

Objects can be created by a transaction or persist in on-chain blockchain storage. The v1 and v2 in Figure 2c are two objects created by the transaction. On-chain objects are created and stored by some transactions. A new transaction can use on-chain objects owned by the transaction sender, or are simply shared objects. On-chain objects are non-“Hot Potato” by definition and thus a fuzzer can directly use them as long as the type is matched.

*Definition 3.2 (Transaction).* A transaction  $S$  in Move is a sequence of function calls without control flows.

On both Sui and Aptos, the two prevailing blockchains that adopt the Move language, a transaction could carry multiple Move function calls with slightly different implementation details. Sui supports *Programmable Transaction Block* (PTBs) composed with a sequence of function calls, each of which can use either an existing object or a value returned from previous Move function calls. Aptos allows users to ad-hoc compile and run a piece of Move code as a transaction, named *Script*, instead of deploying a Move contract, which essentially is a superset of *Programmable Transaction Block* with arbitrary control flows. Both mechanisms enable a transaction to do multiple Move function calls in an atomic fashion. Thus, without loss of generality, transaction in this paper is defined as a sequence of function calls without control flows.

*Definition 3.3 (Object consumption).* For a function call  $f$ , if object  $o$  is part of its input, we say the function  $f$  consumes object  $o$ .

*Definition 3.4 (Object production).* For a function call  $f$ , if object  $o$  is part of its output, we say the function  $f$  produces object  $o$ .

These two definitions are important for us to describe how we handle “Hot Potato” objects during transaction generation and mutation.

#### 3.2 BELOBOG in a Nutshell

BELOBOG is essentially a gray-box fuzzing method. It shares the typical components of the conventional fuzzers, such as seed initialization and seed mutation. We briefly discuss the fuzzing loop and postpone the technical details to the next sections.

- **Step 1: Seed initialization.** Initially, the seed corpus contains no transactions. Since smart contracts are stateful programs, transactions may use on-chain objects. We thus initialize an object pool, denoted as  $\text{Pool}_{obj}$ , by crawling all on-chain objects that are related to the target smart contract.  $\text{Pool}_{obj}$  will be initialized when the fuzzer starts and be reset to the initialized state before generating a new transaction.
- **Step 2: Type graph construction.** Since Move is a strongly typed language, we need to ensure the type validity of all the generated/mutated transactions. To this end, we construct a type graph  $G$  to capture all necessary type-related information, which can then be used to

guide the generation and mutation of transactions. Our usage of the type graph  $G$  ensures that all the transactions produced are well-typed. Technical details will be introduced in Section 3.3.

- **Step 3: Transaction generation and mutation.** In each iteration, BELOBOG either randomly generates a new transaction or mutates an existing seed transaction.
  - *For random generation*, BELOBOG first selects a random function node in the type graph. Then, BELOBOG walks on the type graph to find additional function nodes that are required to make a valid transaction. For each function call, BELOBOG guarantees that each input object is produced by some earlier function or from `Poolobj`, and each output “Hot Potato” object is properly consumed.
  - *For random mutation*, BELOBOG first selects a seed transaction from the seed pool and then mutates it by modifying constant values or adding/removing function calls. Since newly added or removed function calls may break type validity, BELOBOG again uses the type graph to maintain validity.
  - *Concolic execution*. The above generation and mutation processes only care about the type validity of a transaction. For primitive types such as `u64` that are used as function inputs, they all use a random strategy to choose or mutate their values. In fact, hard-coded conditions or checks are prevalent in smart contracts. For example, line 13 in Figure 5a can only be triggered when the inputs satisfy the value constraint, which is nearly impossible to reach with random values. To tackle this challenge, we design and implement a concolic executor based on the Move Virtual Machine.

Technical details about transaction generation and mutation will be introduced in Section 3.4. Technical details about the concolic executor will be introduced in Section 3.5.

- **Step 4: Transaction execution and feedback collection.** The generated new transaction will then be executed on the Move Virtual Machine (MoveVM). If the transaction increases the code coverage<sup>1</sup>, it will be added to the seed corpus for further mutation. If the execution reveals an oracle violation, it will be reported as a potential vulnerability. Different from C/C++ software, which typically has a standard oracle, such as buffer overflow, there is no standard oracle in smart contracts, and the oracle problem remains an unsolved problem [16, 24, 31]. In our implementation, we provide five pre-defined common oracles that we observe in Move smart contracts. Additionally, we expose a user-defined oracle interface that enables the tool users to define their own oracles. Technical details about oracle definition will be discussed in Section 4.

### 3.3 Type Graph Construction

Given a target smart contract, our first step is to construct its type graph  $G = (V, E)$ , where  $V$  is the set of nodes representing either a type (struct or primitive type) or a function definition, and  $E$  is the set of labeled edges. Figure 5a shows an example Move smart contract. It contains three types, *i.e.*, primitive type `u64`, newly created object type `Receipt<T>`, and object type `Coin<T>` from the standard library, and four function definitions. Figure 5b shows the type graph we constructed for this smart contract. There are three different kinds of nodes in a type graph:

- **Default type node:** A type node that is a primitive type, such as `u64`, or a struct type with at least drop or store capability, such as `Coin<T>`.
- **Hot potato type node:** A type node where the type has neither drop nor store ability, such as `Receipt<T>`.
- **Function node:** A function node that represents a function definition, such as `loan<T>()`.

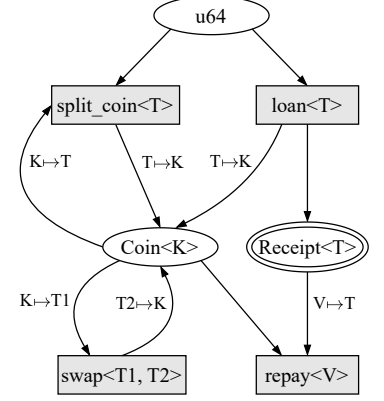
<sup>1</sup>We use the standard branch coverage in our implementation.

```

1  module pool;
2  public struct Receipt<T> {...}
3
4  public fun loan<T>(amount: u64): (Coin<T>, Receipt<T>) {...}
5
6  public fun repay<T>(coin: Coin<T>, receipt: Receipt<T>) {
7    // Function to repay flash loan with coins.
8    let Receipt {
9      amount,
10     fee
11   } = receipt; // unpack the receipt and consume it.
12   let paid = ...// extract from the input coin
13   assert!(paid == amount + fee, "not enough repay");
14   ... // potentially vulnerable path
15 }
16 public fun swap<T1, T2>(coin: Coin<T1>): (Coin<T2>) {...}
17
18 public fun split_coin<T>(coin: Coin<T>, split: u64)
19   : (Coin<T>, Coin<T>) {...}

```

(a) A constructed Move smart contract with two struct definitions and two function definitions.



(b) Our constructed type graph for the code in (a).

Fig. 5. A constructed Move smart contract (left) and the built type graph for it (right).

Move smart contracts usually rely on standard libraries, or third-party smart contracts. Consequently, a smart contract may use types defined somewhere else. In order to build a type graph with complete information, we need to traverse all the libraries and smart contracts that are referenced by our target smart contract. In our implementation, BELOBOG supports building the type graph for a group of smart contracts at the same time.

Our type graph is inspired by the *signature graph* introduced in program synthesis [19] and the *API graph* used in API-driven program synthesis [26]. Our type graph construction initializes the type graph  $G$  with every used primitive type as a default type node  $n_t$ . Then, it analyzes all the struct types. If the struct has at least drop or store ability, it is added as a default type node  $n_t$ , illustrated as ovals in Figure 5b; Otherwise, it is added as a “Hot Potato” type node  $n_h$ , illustrated as double-circled ovals in Figure 5b. Then, for each function definition  $f$  in the target smart contract, we proceed as follows.

- Add  $f$  as a function node in the type graph  $G$ .
- For each input type  $t_i$ ,
  - If  $t_i$  is a non-polymorphic type, add the edge  $t_i \rightarrow f$  that connects the type node  $t_i$  and the function node  $f$  to the type graph  $G$ . For example,  $u64 \rightarrow \text{loan}<T>()$  is added as  $u64$  is the input type of  $\text{loan}<T>()$ .
  - If  $t$  is a polymorphic type with type parameter  $K$ , i.e.,  $t<K>$ , add the edge  $t<K> \xrightarrow{K \mapsto T} f$  to the type graph  $G$ , where the edge connects the type node  $t<K>$  and the function node  $f$ . The annotation  $K \mapsto T$  means type substitution. Since there is no subtyping<sup>2</sup> in Move, type substitution can be greatly simplified. We use  $K \mapsto T$  to represent that  $T$  will be instantiated with the concrete type that instantiates  $K$ . For example,  $\text{Coin}<K> \xrightarrow{K \mapsto T} \text{split\_coin}<T>$  is added according to this rule. During transaction generation, when we instantiate  $\text{Coin}<K>$

<sup>2</sup>Although tuple in Move supports a very limited subtyping rule, it cannot be used as type parameter [7].

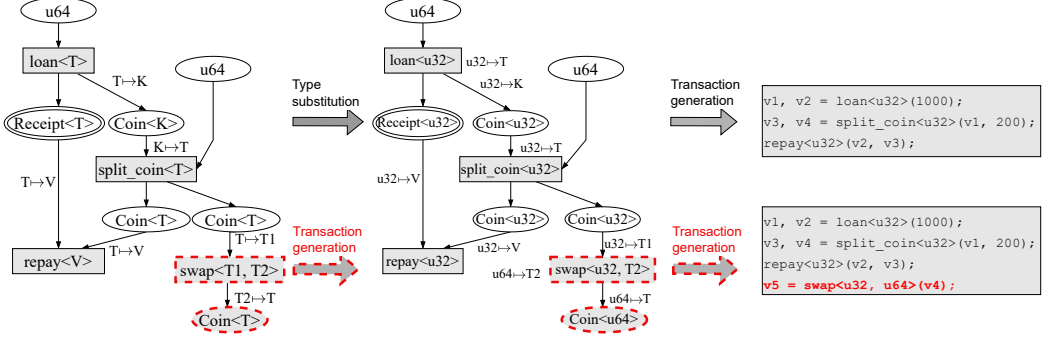


Fig. 6. Illustrative transaction generation starting from a graph trace (left), then concretizing type parameters by type substitution (middle), and finally instantiating a transaction (right). The **dashed red nodes** demonstrate the mutation process of graph trace extension.

as  $\text{Coin}\langle u64 \rangle$  and pass it as input to  $\text{split\_coin}\langle T \rangle()$ , the function  $\text{split\_coin}\langle T \rangle()$  also instantiates  $\text{ints}$  type parameter  $T$  to  $u64$ .

- For each output type  $t_o$ , similar to the input type, we add the edge and the annotation depending on whether it is a non-polymorphic or a polymorphic type.

This complete type graph will enable us to know which type can be produced or consumed by which functions, thus facilitating our later transaction generation and mutation.

### 3.4 Transaction Generation and Mutation

There are two ways of producing a transaction in BELOBOG, namely generation and mutation. They share the same high-level idea but differ in the implementations. Below we introduce the details.

**Transaction Generation.** The high-level idea of generating a transaction is to search for a graph trace  $G_{\text{trace}}$  in the type graph  $G$  such that we can generate a valid transaction by instantiating  $G_{\text{trace}}$ . It has three main steps.

- **Step 1: Graph trace construction.** The purpose is to walk on the type graph to find a graph trace  $G_{\text{trace}}$ , whose edge nodes are all *default type nodes*. Theoretically, we could start from any node in  $G$  to find such a trace. But to make our algorithm run faster, we start from a function definition that only accepts void or primitive types as input. This choice does not affect the overall outputs of the algorithm. The reason is that, unless using shared objects from  $\text{Pool}_{obj}$ , the first function call in a valid transaction must create objects with void or primitive types. For example, the figure on the left in Figure 6 shows one graph trace obtained by walking on the example type graph in Figure 5b starting from  $\text{loan}\langle T \rangle$ .
- **Step 2: Type substitution.** Since  $\text{Move}$  accepts type parameters, before generating the final transaction, we need to find the concrete types for each type parameter in the graph trace. Since types in a  $\text{Move}$  smart contract are usually limited (fewer than 10), we use a heuristic-based method to iteratively find suitable types for each type parameter. For example, the middle figure in Figure 6 shows the graph traces after type substitution.
- **Step 3: Transaction generation.** With a concretized graph trace, we can then generate the final transaction by instantiating it. The process starts from all type nodes without incoming edges in the graph trace and adopts a width-first enumeration to generate each function call. For example, in Figure 6, we start from the type node  $u64$  and generate the function call  $\text{loan}\langle u32 \rangle(1000)$ , where 1000 is a random value generated according to its type. Then we

generate the `split_coin<u32>()` and `repay<u32>()` function calls to consume all objects created by `loan<u32>()`.

The above process can be repeated multiple times, creating a new graph trace each time. Even for the same starting node, the algorithm can generate multiple graph traces, while each trace can be instantiated differently when using different concrete types to substitute type parameters. Note that for primitive types, such as `u64`, we use random values for them when generating a transaction, such as `1000` for `loan<T>()` in Figure 6. Although using random values is a common practice in fuzzing, it can be very hard for them to reach certain code path that requires strict condition checks. To overcome this limitation, we design a concolic executor in BELOBOG. We will introduce more details in Section 3.5.

**Transaction Mutation.** Transactions that increase code coverage will be added to the seed corpus for further mutation. When a transaction  $S$  is selected for mutation, we first recover its graph trace  $G_{trace}$  and use one or several of the following mutators to produce a new transaction.

- **Mutator 1: Value mutation.** For primitive values in  $S$ , we mutate them by doing AFL-style havoc mutation[12], *i.e.*, randomly mutating several bits or bytes in  $S$ .
- **Mutator 2: Graph trace extension.** Randomly select a type node in  $G_{trace}$  that is not consumed and extend the graph trace by walking on the type graph  $G$  until we get a new valid graph trace  $G'_{trace}$ , then instantiate  $G'_{trace}$  to a transaction. For example, the dashed nodes in Figure 6 illustrate this mutation. We first select the type node `Coin<T>` that is not consumed yet in the original  $G_{trace}$ . Then we walk on the type graph  $G$  to add the function node `swap<T1, T2>()` to the  $G_{trace}$ . We will then do type substitution on it and finally instantiate a mutated transaction from it. Note that it is possible that there is no such type node in  $G_{trace}$ . In this case, this mutator will be skipped.
- **Mutator 3: Function call insertion.** Randomly select a new function node  $f'$  in the type graph and construct a graph trace  $G'_{trace}$  by trying to connect  $f'$  to  $G_{trace}$ . We select a type node in  $G_{trace}$  that is not consumed as the target, and then try to search for the connection path by random walk. If this search process succeeds, this path will be added to  $G_{trace}$ , and then we will include additional necessary nodes to make this new graph trace complete. If this search process fails, meaning that there is no valid type node in the original  $G_{trace}$  or there is no such a path, we invoke the *transaction generation* algorithm with  $f'$  as the starting node to generate a transaction. This transaction will be appended to the original transaction to get a new transaction. For example, the dashed nodes in Figure 6 can also illustrate this process if we view `swap<T1, T2>` as the selected function node.
- **Mutator 4: Function call removal.** Randomly select a function node in  $G_{trace}$  and then remove it from  $G_{trace}$ . This removal may break the validity of  $G_{trace}$ , and thus we keep removing necessary nodes until the remaining nodes constitute a valid graph trace  $G'_{trace}$ . Finally, we instantiate  $G'_{trace}$  into a new transaction.

For a seed transaction  $S$  and its graph trace  $G_{trace}$ , more than one of the mutators may be applied. If this resulting transaction  $S'$  triggers new code coverage,  $S'$  will be added into the seed corpus for further mutation.

### 3.5 Concolic Execution

Concolic executor is an important component for ensuring BELOBOG reach critical paths in the target Move smart contracts. We implement the concolic executor on top of the MoveVM, which provides the necessary toolchains for us to obtain the runtime values of variables and stacks. In particular, stack values in MoveVM are also strongly typed, making it relatively easy for us to symbolically model vectors, which is notoriously hard for other languages such as C/C++ [15].

For a transaction, the general process of performing concolic execution on it consists of two steps: *constraint collection* and *concolic mutation*.

**Constraint collection.** We trace the execution of a transaction in its MoveVM bytecode and collect all the constraints along the execution path. For each function input that has a primitive type, BELOBOG attaches a symbolic value to it; For a vector of primitive type, BELOBOG attaches a vector of symbolic values to model its elements correspondingly. There are three major kinds of constraints that will be collected during execution, as detailed below.

- **Arithmetic operations.** We collect arithmetic constraints on arithmetic operations, such as *add* and *mul*. In Move, such operations will abort if an overflow happens. Thus, BELOBOG will also include additional constraints to ensure that the solved values will never overflow.
- **Cast operations.** Due to the type requirement, operands of an arithmetic operation need to be of the same type. MoveVM uses explicit cast bytecode, such as *castu16*, to change the operand type accordingly. At runtime, if a value exceeds the range of the target type, MoveVM will abort the execution. For example, if we cast a variable of type *u64* to *u32*, it is abnormal if the runtime value of the variable exceeds the maximum value of *u32*. BELOBOG encode value constraints for cast operations by observing the source and destination types.
- **Vector access.** When a transaction accesses a vector, the MoveVM will abort the execution if the access is out of the vector's bounds. To avoid generating invalid values, BELOBOG encodes the index constraints for vector accesses.

**Concolic mutation.** Once BELOBOG collects all the constraints of a transaction, it will randomly pick one or more branches and flip their conditions in the constraints. The mutated constraints will be sent to an SMT solver. If a new set of inputs is found by the solver, the corresponding values in the original transaction will be replaced accordingly to obtain a new transaction. This new transaction will then be executed and added to the seed corpus if it increases code coverage.

We take the `repay<T>()` function in Figure 5a to illustrate how our concolic execution works and why it is necessary. Assume that the transaction is the one shown on the top right of Figure 6. The vulnerable path will only be explored if the assertion check `paid==amount+fee` in line 13 is satisfied. Our symbolic executor will make the first argument of `loan<T>()` and the second argument of `split_coin<T>()` symbolic and hold the symbolic values `SYM1` and `SYM2`, respectively. The symbolic executor will track the propagation of these symbolic values. Suppose that the `paid=SYM2`, `amount = SYM1`, and `fee = 0.001*SYM1` in line 13. To make the assertion pass, we need to satisfy the constraint `SYM2==SYM1+0.001*SYM1`. Our concolic executor can easily find the valid input values by assigning `SYM2=1001` and `SYM1=1000`. However, the chance of getting valid input values would be negligible without concolic execution.

## 4 Implementation

We build BELOBOG from scratch, totaling 17k lines of Rust code. Specifically, we spend 4.1k lines to support forking a specific on-chain checkpoint to enable fast and local on-chain data access. With this support, BELOBOG can initialize the object pool with on-chain objects. The type graph building, graph tracing, and transaction generation and mutation take 3.7k lines, while the concolic mutation takes 2k lines.

**Oracle.** In order to detect bugs in Move smart contracts, we need oracles to decide if each transaction is executed correctly. Unfortunately, designing an automated and general oracle for fuzzing smart contracts has been a long-standing and unsolved problem [31]. The main reason is that oracles often depend on the business model or semantics of the target smart contract. One can think of it as detecting functional bugs in general software, which remains unsolved for decades. Therefore,

following the similar practice in previous smart contract fuzzers like Foundry[6] and Echidna[14], BELOBOG provides complementary Move libraries to enable users to implement custom oracles in pure Move language by emitting specific events when invariants are violated. We will introduce more details in the evaluation in Section 5.

A recent empirical study [25] defines some common oracles for Move smart contracts. These oracles are mostly for checking mild issues in a smart contract, such as *unnecessary type conversion*, which may waste some gas. We implement all five oracles from [25] that can be used during fuzzing in BELOBOG. Additionally, we implement a new general oracle “*earning profits*” for detecting no-cost earnings. Below is a list of the details about these oracles.

- *Infinite Loop*: detect infinite loops in a contract. BELOBOG checks if the expression and value of a branch condition remain unchanged until the execution is out of gas.
- *Precision Loss*: check both operands of an arithmetic operation to determine if precision loss happens, e.g.,  $5/2 \Rightarrow 2$  instead of  $2.5$ .
- *Unnecessary Type Conversion*: check if the type of the stack top during execution is the same as the cast destination type.
- *Unnecessary Bool Judgment*: check if the operands of *EQ* or *NEQ* is a constant boolean value. In this case, the comparison is not necessary because the other operand itself could be used to determine the condition by either *BrFalse* or *BrTrue*.
- *SHL Overflow*: detect left shift overflow.
- *Earning Profits*: check if the amount of coins owned by the transaction sender increased after the execution. Since we do not add additional coins in generated transactions, the amount of coins should never be increased. Otherwise, we can earn profits without any cost.

Note that although we implement all these predefined oracles, we view them as the default oracles when users do not provide a better oracle. In our evaluation, we manually write oracles for the target smart contracts for a realistic experimental setup.

**Supporting different blockchains.** To date, the codebases and semantics of Sui and Aptos, two prevailing blockchains using Move, have diverged significantly. This leads to distinct features with entirely different interfaces. Thanks to the flexible and modular design of BELOBOG, we can easily support both blockchains with minor efforts. Thus, we believe our implementation can also be easily extended to other blockchains. For different blockchains, most changes are in coverage instrumentation and instruction tracing, totaling 483 and 638 lines of difference in the two blockchains, respectively.

## 5 Evaluation

In this section, we evaluate BELOBOG on datasets collected from real-world auditing and compare it with other related fuzzers. We aim to answer the following research questions in this section.

- **RQ1 (Bug finding).** *Is BELOBOG effective in finding vulnerabilities in real-world Move smart contracts?*
- **RQ2 (Coverage and throughput).** *Can BELOBOG achieve higher code coverage and throughput?*
- **RQ3 (Abalation study).** *How do the key components in BELOBOG affect the overall effectiveness?*
- **RQ4 (Case study).** *Can BELOBOG reproduce real-world incidents?*

**Dataset Construction.** There are two datasets used in our evaluation. First dataset, **DS1**, is assembled from the audit reports provided by a well-known Web3 auditing company, BitsLab<sup>3</sup>. In this dataset, all Move smart contracts are audited manually by human experts with detailed

<sup>3</sup>The company shared the anonymized data with us.

vulnerability information. These reports provide us with ground-truth vulnerability locations to validate the bug-finding capability of fuzzers. Although it is possible that some vulnerabilities escape human experts’ audit, it does not affect the evaluation result of our experiments. Each vulnerability is labelled based on its severity. We select 37 projects that contain at least one *Medium* vulnerability in the past year. This yields 960 smart contracts with 9 *Critical*, 43 *Major* and 35 *Medium* vulnerabilities.<sup>4</sup> *Critical* vulnerabilities mostly lead to the drain of liquidity or shares. *Major* vulnerabilities usually relate to the design flaws of the business models or improper access control that are not directly exploitable. *Medium* severity usually indicates a potentially incorrect setup or configuration causing minor losses or unexpected results, *i.e.*, non-security issues. We exclude the *Minor* and *Informational* vulnerabilities because they are either code style issues or harmless defects, such as inconsistent events emitted or gas waste, and the audit reports mark these issues as not compulsory to fix. We attach the full anonymized contents of **DS1** in our supplementary material. The second dataset, **DS2**, is from MOVESCAN [25]. It contains 72 open-sourced Move projects, totalling 244 smart contracts.<sup>5</sup>

For **DS1**, since we have the ground-truth audit reports, we manually implement the oracle for each smart contract. Our oracle implementations are easy, averaging 15 lines per vulnerability. Since all projects in **DS1** are deployed on Sui, our evaluation on **DS1** only uses the Sui blockchain. For **DS2**, we enable all our predefined oracles and evaluate on both Sui and Aptos blockchains. Since the choice of blockchain does not affect the effectiveness of fuzzers, we do not explicitly separate them in our discussion.

**Baseline Fuzzers.** We initially hoped to directly adopt two most relevant and state-of-the-art smart contract fuzzers, ITrYFUZZ [24] and SUI-FUZZER [3], as the baselines. Unfortunately, at the time of writing, there have been 6 months and 19 months since the last maintenance of ITrYFUZZ and SUI-FUZZER, respectively. Our preliminary efforts showed that they failed to start their fuzzing process in most smart contracts. In order to evaluate them, we have to fix these two fuzzers before conducting the evaluation. Specifically, we add another 503 lines for ITrYFUZZ to upgrade its Sui dependency, resolve various dirty hacks that result in MoveVM invariant violations, and fix a few bugs causing segmentation faults and dead loops. For SUI-FUZZER, we also upgrade its Sui dependency, fix its building process, and extend its support for the `init` function executed during Move contract deployment by another 381 lines of code.

**Experiment Setup.** We conduct all the experiments on two machines, both equipped with an EPYC 7B13 processor and 1 TiB memory and running Ubuntu Server 24.04. We also bind each fuzzing campaign to a specific logical core and use a *tmpfs* memory partition for any fuzzer output to avoid interference. We repeat each fuzzing campaign five times, each for 12 hours.

## 5.1 RQ1: Bug Finding

**Bug-finding results.** Table 1 shows the bug-finding results of different fuzzers on the **DS1**. BELOG demonstrates the strongest bug-finding capability in all projects. Out of all 109 bugs, BELOG successfully detects more than 85% (93) of them. Notably, BELOG detects *all the 9 Critical bugs*, the most concerning vulnerabilities in the real world. On 24 out of 37 projects, BELOG finds all the bugs. In comparison, ITrYFUZZ only detects 3 bugs, with all of them being *Major* or *Medium*, and no *Critical* bugs can be found.

Regarding **DS2**, since it is created by a static analysis tool MOVESCAN, we only compare our tool with it. In total, BELOG finds 181 defects, while MOVESCAN reveals 176 defects with 20 false

<sup>4</sup>Following MOVESCAN convention, we deem a Move module as a smart contract and one Move project usually contains multiple Move modules.

<sup>5</sup>We exclude 1 incorrectly labeled project, 1 project that deletes its repository, and 3 projects that no longer build.

Table 1. Overall bug-finding results on **DS1**. Per-smart contract comparison between BELOBOG and ItyFuzz. "MA" and "ME" is abbreviated for "Major" and "Medium", respectively.

| Target   | BELOBOG |          |       |        |      | ItyFuzz | Target       | BELOBOG   |          |           |           |                | ItyFuzz |
|----------|---------|----------|-------|--------|------|---------|--------------|-----------|----------|-----------|-----------|----------------|---------|
|          | Total   | Critical | Major | Medium |      |         |              | Total     | Critical | Major     | Medium    |                |         |
| 8bc50340 | 5/5     | -        | 2/2   | 3/3    | 0    |         | 8f0c3e4c     | 5/6       | -        | 0/1       | 5/5       | 0              |         |
| c955a990 | 4/4     | 2/2      | 2/2   | -      | 0    |         | d46b85fc     | 1/2       | 1/1      | -         | 0/1       | 0              |         |
| 902aaf72 | 2/2     | -        | 1/1   | 1/1    | 0    |         | c40e8d17     | 0/1       | -        | -         | 0/1       | 0              |         |
| 1e6f57ad | 4/4     | -        | 1/1   | 3/3    | 0    |         | 4a93535d     | 1/2       | -        | -         | 1/2       | 1 ME (FP)      |         |
| 916a5d9c | 3/4     | -        | 0/1   | 3/3    | 0    |         | f1b4dc69     | 2/3       | -        | 2/3       | -         | 0              |         |
| 73c07115 | 0/2     | -        | 0/2   | -      | 0    |         | 1b894d8a     | 2/2       | -        | -         | 2/2       | 0              |         |
| c52c5116 | 1/2     | -        | 0/1   | 1/1    | 0    |         | 60cb5d6d     | 3/5       | 1/1      | 2/3       | 0/1       | 0              |         |
| d0c0ef13 | 4/4     | -        | 1/1   | 3/3    | 0    |         | b3e614c7     | 2/2       | -        | 1/1       | 1/1       | 0              |         |
| e3564d47 | 1/1     | -        | 1/1   | -      | 0    |         | 7d85e770     | 2/3       | -        | 1/2       | 1/1       | 0              |         |
| e8a96a03 | 2/2     | -        | 1/1   | 1/1    | 0    |         | c2ade084     | 2/2       | -        | 2/2       | -         | 0              |         |
| d834970c | 2/2     | 1/1      | 1/1   | -      | 0    |         | b830405a     | 3/4       | -        | 1/2       | 2/2       | 0              |         |
| 074ef533 | 6/6     | 2/2      | 2/2   | 2/2    | 0    |         | c74a124c     | 4/4       | -        | 3/3       | 1/1       | 0              |         |
| 52df0499 | 4/4     | -        | 2/2   | 2/2    | 1 MA |         | d492c273     | 1/2       | 1/1      | -         | 0/1       | 0              |         |
| 79451d4e | 3/3     | 1/1      | 1/1   | 1/1    | 0    |         | d3d51621     | 1/1       | -        | 1/1       | -         | 0              |         |
| 487e891e | 3/3     | -        | 2/2   | 1/1    | 0    |         | 67462d31     | 1/2       | -        | 1/2       | -         | 0              |         |
| d3b5b821 | 2/3     | -        | 0/1   | 2/2    | 0    |         | fb8888cb     | 2/2       | -        | 2/2       | -         | 0              |         |
| 415b8e9d | 5/5     | -        | 3/3   | 2/2    | 0    |         | 79d6665b     | 3/3       | -        | 3/3       | -         | 0              |         |
| 6b14ac4c | 3/3     | -        | 1/1   | 2/2    | 0    |         | 54197c4b     | 3/3       | -        | 2/2       | 1/1       | 0              |         |
| 54995d2b | 1/1     | -        | -     | 1/1    | 1 ME |         |              |           |          |           |           |                |         |
|          |         |          |       |        |      |         | <b>Total</b> | <b>93</b> | <b>9</b> | <b>42</b> | <b>42</b> | <b>3 (1FP)</b> |         |

positives. The further breakdown of our predefined oracles and smart contracts on **DS2** is available in our complementary appendix.

**SUI-FUZZER Deficiency.** We do not include the result of SUI-FUZZER in the table because it does not yield any findings even after we fix many bugs in it. SUI-FUZZER always aborts early in our experiments because it does not support shared objects. We then investigate that even if SUI-FUZZER could support shared objects, it would still fail all the fuzzing experiments because it still has several inherent design flaws: (1) it will forge objects of any types by mutating the objects unconditionally, leading to massive amount of false positives; (2) it only supports testing a single Move smart contract with selected functions while the real-world defects mostly span multiple ones; (3) it only supports filling up objects from the existing global owned objects, making it unable to correctly handle “Hot Potato” objects; and (4) it does not consider types of object and usually fill in objects with unmatched types.

**Case study in DS1.** We select one *Major* vulnerability, “Incorrect reserve calculation”, from the project *8bc50340* of **DS1** to illustrate the design of our custom oracle and the effectiveness of our approach.<sup>6</sup> In general, a custom Move oracle of BELOBOG is designed to be a thin wrapper for each target function by logging relevant values and checking invariants. Figure 7 illustrates a simplified version of the custom oracle and original contract, where function `oracle_buy<T>()` is the manually written oracle for function `buy<T>()`. This smart contract provides a function

<sup>6</sup>The code and description are also anonymized.

```

1  pub struct SUI has drop { ... } // struct representing the standard SUI coin
2
3  public fun get_reserve(): u64 { ... }
4  public fun buy<T>(payment: Coin<SUI>, amount: u64): (Coin<SUI>, Coin<T>) { ... }
5
6  /// Our oracle wrappers from the module "oracle"
7  public fun oracle_buy<T>(coin: Coin<SUI>, amount: u64): (Coin<SUI>, Coin<T>) {
8      let previous_reserve = get_reserve();
9      let out = buy(item_id); // Forward the parameters to the original implementation
10     if (previous_reserve - get_reserve() != amount) {
11         report_oracle_violation("Incorrect reserve calculation.");
12     }
13     out
14 }

```

Fig. 7. An example oracle for detecting a *Major* vulnerability.

```

1  fun count_leading_zeros(x: u16) {
2      if (x & 0xFF00 == 0) { x = x << 8; };
3      ...
4  }

```

Fig. 8. A spurious left shift overflow MOVESCAN reports

buy<T>() to buy <T> with the <SUI> coins, a native currency provided by the standard library. The oracle function oracle\_buy<T>() serves as a thin wrapper for buy<T>(), forwarding the function call while verifying that the reserve is appropriately decreased in line 10. When a transaction is to be executed, all of the function calls to buy<T>() will be replaced by oracle\_buy<T>() to check oracle violations during the execution of this transaction. With this oracle, BELOBOG can detect this vulnerability in a few seconds once the input coin matches other constraints in the buy<T>() function. However, even with this oracle, IryFUZZ cannot generate a valid function call to buy<T>() due to not supporting the type parameters.

**False positives in MOVESCAN.** MOVESCAN has 20 false positives while BELOBOG has no false positives. We take the Figure 8 as an example to illustrate how MOVESCAN produces false positives. Since MOVESCAN is a static analyzer, it does not know the actual value of each parameter and thus it will assume the maximum value for each variable to detect overflow. In this case, MOVESCAN assumes  $x == 65535$  and identifies  $x = x << 8$  as a left shift overflow but ignores the guard  $x \& 0xFF00 == 0$ , leading to a false positive. BELOBOG can always check the arithmetic at runtime and thus avoid any such false positives.

**False negatives in BELOBOG.** While BELOBOG shows strong capability of finding bugs on DS1, we still miss 15% (16) bugs for two reasons. First, some bugs are out of scope. For instance, some contracts are vulnerable to Denial-of-Service attacks because of iterating over too large vectors, while some other vulnerabilities relate to incorrect logs during execution. As a smart contract fuzzer, BELOBOG is not capable of detecting such vulnerabilities that do not affect the integrity of smart contracts. Second, some bugs need administrator privileges to trigger. For example, centralization risk is a common vulnerability where the administrator has too much power. Both BELOBOG and the baselines IryFUZZ and SUI-FUZZER do not assume any special capabilities for testing. These bugs go beyond the capability of a dynamic fuzzer, and we consider them as one of our limitations. We will discuss more limitations in Section 6.

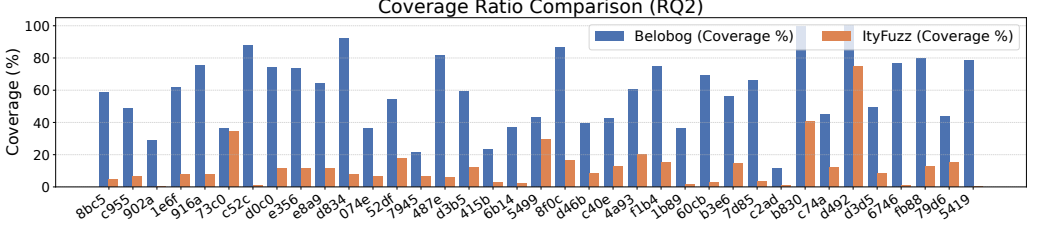


Fig. 9. BELOBOG vs. ITYFUZZ in terms of basic-block coverage on all projects from **DS1**. Each point represents one project. BELOBOG consistently achieves much higher coverage than ITYFUZZ, indicating that it can explore deeper program states and reach more code.

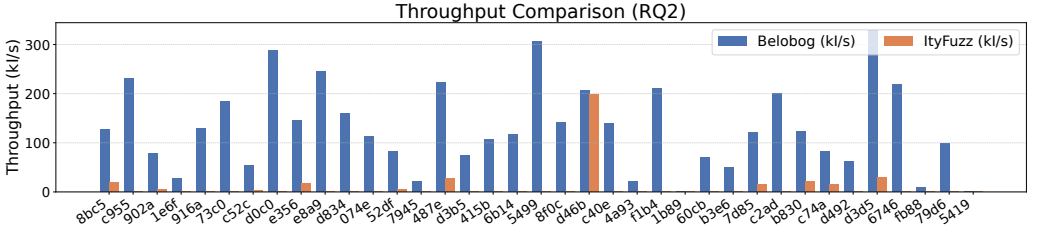


Fig. 10. Fuzzing throughput of BELOBOG and ITYFUZZ on all projects from **DS1**. Throughput is measured in thousands of executed instructions per second (kl/s). BELOBOG executes significantly more instructions per second than ITYFUZZ across all targets.

## 5.2 RQ2: Coverage and Throughput

We collect the basic-block coverage of each smart contract during fuzzing. Figure 9 shows the averaged results in each smart contract. Across all 67 smart contracts in **DS2** and **DS1**, BELOBOG reaches significantly higher coverage than ITYFUZZ on all of them. Specifically, BELOBOG reaches an average of 55.6% coverage, while ITYFUZZ only reaches 8.8%.

Figure 10 shows the throughput of BELOBOG and ITYFUZZ. BELOBOG has an average of 129K transaction executions per second, while ITYFUZZ has an average of 10K transaction executions per second. BELOBOG achieves much higher throughput than ITYFUZZ in nearly all of the smart contracts, except for the smart contract d46b, on which both fuzzers achieve similar throughput. The core reason is that this smart contract is relatively simple, without many type parameters and “Hot Potato” object types and thus ITYFUZZ is easier to find objects of expected types to build seeds. The overall high throughput of BELOBOG is because we optimized our implementation to incrementally build transactions during fuzzing, avoiding redundant transaction builds. However, ITYFUZZ implements a snapshot mechanism, which greatly suffers from the state explosion.

## 5.3 RQ3: Ablation Study

In order to understand the impact of individual components in BELOBOG, we additionally implement three variants by excluding a few components and evaluate their bug-finding capabilities on **DS1**.

- **BELOBOG-NTG**: In this variant, we no longer generate the type graph but instead use a random havoc mutation like ITYFUZZ.
- **BELOBOG-NTY**: In this variant, we do not fill type parameters.

```

1  public fun mul_div(a: u128, b: u128, c:u128): u128 {
2      a * b / c
3  }
4  public fun compute_boost_factor(vestar_amount: u128, farm_amount: u128, total: u128): u64 {
5      // precision loss because division before multiplication
6      let dividend = mul_div(vestar_amount, FACTOR * 3, total) * FACTOR;
7      let divisor = mul_div(farm_amount, FACTOR * 2, total);
8      ((dividend / divisor) + FACTOR) / (FACTOR / 100)
9  }
10 public fun calculate_boost_weight(amount: u128, boost_factor: u64): u128 {
11     // precision loss because division before multiplication
12     amount * (boost_factor as u128) / BOOST_FACTOR_PRECESION
13 }
14 public fun update_boost_factor(...) {
15     let boost_factor = compute_boost_factor(...);
16     let weight = calculate_boost_weight(asset_amount, boost_factor);
17     ...
18 }

```

Fig. 11. A Precision Loss defect of the *starswap-core-aptos* project.

Table 2. Bug-finding results of BELOBOG and its variants on **DS1**. GroundTruth denotes the number of ground-truth bugs.

| Category | GroundTruth | BELOBOG | BELOBOG-NCE | BELOBOG-NTG | BELOBOG-NTY | BELOBOG-DUMB |
|----------|-------------|---------|-------------|-------------|-------------|--------------|
| Critical | 9           | 9       | 5           | 0           | 4           | 0            |
| Major    | 53          | 42      | 28          | 2           | 26          | 1            |
| Medium   | 47          | 42      | 33          | 1           | 28          | 1            |
| Total    | 109         | 93      | 66          | 3           | 58          | 2            |

- **BELOBOG-NCE:** In this variant, we remove the concolic executor from BELOBOG.
- **BELOBOG-DUMB:** In this variant, we remove concolic execution, type graphs, and type parameter support.

The last column in Table 2 shows the result of removing all three key components in BELOBOG. Only two bugs can be detected, suggesting the criticality of the three components. Below we discuss the results of each variant.

**Impact of type graph.** The **BELOBOG-NTG** column in Table 2 shows the bug-finding result after removing the type graph. Compared to our default BELOBOG, *none of the Critical* bugs can be detected, and the overall bug detection ratio drops from 85% to nearly zero. The main reason is that without type graphs, **BELOBOG-NTG** can no longer guarantee that “Hot Potato” objects are consumed properly and thus many functions can not be covered.

**Impact of type parameters.** The **BELOBOG-NTY** column in Table 2 shows the bug-finding result after removing the support of type parameter. Compared to our default BELOBOG, only 4 *Critical* bugs can be detected, and the overall bug detection ratio drops from 85% to 53%. The reason for the bad performance is that 37% of the functions in **DS1** have at least one type parameter and thus all transactions generated by **BELOBOG-NTY** can not make a valid call to them.

**Impact of concolic executor.** The **BELOBOG-NCE** column in Table 2 shows the bug-finding result after removing the concolic executor. Compared to our default BELOBOG, only 5 *Critical* bugs can be detected, and the overall bug detection ratio drops from 85% to 60%. This suggests that concolic execution is crucial to exploring critical paths in smart contracts.

```

1  fun check_tick_range(lower: i32, upper: i32, tick_spacing: u32) {
2      assert!(lower < upper &&
3          lower >= MIN_TICK && upper <= MAX_TICK &&
4          lower % tick_spacing == 0 && upper % tick_space == 0);
5  }
6  public fun create_position(ticker_lower: u64, tick_upper: u64, ...): Position {
7      check_tick_range(tick_lower, tick_upper, TICK_SPACING);
8      ..
9  }

```

Fig. 12. A check from the smart contract *e3564d47*.

## 5.4 Case Study

We provide additional cases from **DS1** to demonstrate the capability of BELOBOG.

**Case 1.** Figure 12 shows an example smart contract derived from *e3564d47* to illustrate how our concolic execution digs into the deep state of a function. The public function `create_position()` internally will check against the parameters with `check_tick_range()`, which further creates a rather tight bound on the input value `tick_lower` and `tick_upper`. ItyFuzz will never reach the code after the `check_tick_range()` because the chance of satisfying the constraints by random mutation is negligible, while BELOBOG can easily pass the assertion with our concolic execution.

**Case 2.** Compared to MOVESCAN, BELOBOG covers every finding it reports with no false positives. We additionally find 27 *Precision Loss*. We attribute such advantages to two reasons: (1) MOVESCAN scans these vulnerabilities within a single function context statically, while BELOBOG is capable of tracking values and invariants across multiple call boundaries at runtime; (2) the concolic execution of BELOBOG can provide the precise context for oracles like *Precision Loss*, *Infinite Loop* and *Unnecessary Bool Judgement*.

We demonstrate this with two *Precision Loss* vulnerabilities from the same smart contract from the *starswap-core-aptos* smart contract of **DS2** on the Aptos chain. The first *Precision Loss* happens at Line 6 in Figure 11 because the division of `total` happens before the multiplication of `FACTOR`. The second *Precision Loss* is at Line 12 in Figure 11, where the division of `FACTOR` to `boost_factor()` happens in the function `compute_boost_factor()` before the multiplication of `amount` in `calculate_boost_weight()` when calling the function `update_boost_factor()`. MOVESCAN misses both vulnerabilities because it only detects within a single function call context and does not trace values across function boundaries. In addition, it only supports a very limited fixed pattern like  $(a / b) * c$  with exactly three operands. BELOBOG finds both *Precision Loss* vulnerabilities because we track symbolic expressions for each value across functions and accurately identify potential precision loss.

## 5.5 RQ4: Real-world Incident Study

In order to understand the real-world vulnerability detection capability of BELOBOG, we select two recent catastrophic incidents on the Sui blockchain, known as *Cetus* and *Nemo*, which resulted in more than \$200 million and \$2.6 million financial loss, respectively.

**Cetus Incident Analysis.** Figure 13 presents a simplified *Cetus* smart contract. Although most arithmetic overflows are properly checked by the MoveVM, left shift is not. Thus, there is a left shift check function `checked_shlw()` implemented in the *Cetus* to prevent such overflows. However, the function incorrectly encodes the left shift overflow threshold as  $v > (0xFFFFFFFFFFFFFFFF << 192)$ ,

```

1  fn checked_shlw(v: u256): u256 { /// private function for implementing left shift overflow check
2      if (v > (0xFFFFFFFFFFFFFFFF << 192)) { // incorrect check with too big value
3          (0, true)
4      } else {
5          (arg0 << 64, false)
6      }
7  }
8  fn get_delta(liquidity: u128, price_low: u128, price_high: u128): u64 { /// private function
9      let price_delta: u128 = price_high - price_low;
10     let prod: u256 = liquidity * price_delta;
11     let (shifted, overflow) = checked_shlw(prod);
12     if (overflow) { abort; } // abort if a left shift overflow is detected
13     (shifted / (price_low * price_high)) as u64 // type cast from u128 to u64.
14         // MoveVM would abort if the u128 value is too big for u64.
15 }
16 public struct AddLiquidityReceipt<T> { /// "Hot Potato" type
17     repay_amount: u64
18 }
19 public fn add_liquidity<T>(amount: u64): AddLiquidityReceipt<T> {
20     let (price_low, price_high) = ...;
21     let repay_amount = get_delta(amount, price_low, price_high);
22     AddLiquidityReceipt { repay_amount }
23 }
24 public fn repay_liquidity<T>(coin: Coin<T>, receipt: AddLiquidityReceipt<T>) {
25     let { repay_amount } = receipt;
26     assert!(coin.value == repay_amount);
27 }

```

Fig. 13. The simplified smart contract from the buggy *Cetus*.

which is much bigger than it is supposed to be, i.e.,  $v \geq (1 \ll 192)$  as the later patch shows. Therefore, any values within the two boundaries, i.e.,  $(1 \ll 192) < v \leq (0xFFFFFFFFFFFFFFFF \ll 192)$ , will lead to an overflowed value be returned.

The real-world exploit constructed a transaction with the call sequence

$$\text{add\_liquidity}(\text{AMOUNT}) \Rightarrow \text{repay\_liquidity}(\dots)$$

with a carefully constructed value AMOUNT. The `add_liquidity()` calls `get_delta()` to calculate the tokens (money) that need to be paid back later, and `get_delta()` calls `checked_shlw()` to avoid too large values. However, a carefully crafted large value AMOUNT can escape the overflow check in `checked_shlw()`, leading to a much smaller `repay_amount` in line 21. In other words, the transaction sender can drain a considerable amount of assets with little cost.

**Fuzzing *Cetus* with BELOBOG.** As shown in Figure 13, the two functions, `add_liquidity()` and `repay_liquidity()`, have to respectively produce and consume a “Hot Potato” object that has the type `AddLiquidityReceipt<T>`. Therefore, with our built type graph, BELOBOG is capable of generating the expected function call sequence. However, the type graph itself alone is not enough to trigger this vulnerability for two reasons: (1) triggering the overflow within `checked_shlw()` is not trivial because the last line of `get_delta()` casts the value from `u256` to `u64`, resulting in a much tighter bound. Our calculation reveals that the chance of randomly mutating a value to trigger a valid overflow is as small as  $2^{-30}$ ; (2) line 26 in `repay_liquidity()` enforces that the input coin has to be equal to the `repay_amount` calculated in line 21. These constraints make it extremely challenging to generate a successful exploit. Fortunately, BELOBOG can easily resolve both obstacles with its concolic executor.

```

1 fun update_price(v: u64): u64 { ... }
2 fun get_price(): u64 { ... }
3 fun current_price(price: u64): u64 {
4     // Unexpected price updating
5     update_price(price);
6     get_price()
7 }
8
9 public fun calculate_amount_by_price(
10     amount: u64, price: u64
11 ): u64 {
12     // Trigger the price update
13     let price = current_price(price);
14     ...
15 }

```

(a) The root cause of *Nemo*.

```

1 public struct Position has store { ... }
2 public struct FlashPosition<T> { ... }
3 public struct Oracle { ... }
4
5 public fun swap<T0, T1>(
6     position: Position, oracle: Oracle
7 ): Position { calculate_amount_by_price(...) }
8 public fun borrow<T>(
9     position: Position, amount: u64
10 ): (Position, FlashPosition<T>) { ... }
11 public func repay<T>(
12     position: Position, receipt: FlashPosition<T>
13 ) { ... }
14 public func get_oracle(...): Oracle { ... }

```

(b) Simplified smart contract of *Nemo* where exploits interacts.

Fig. 14. Simplified source code of *Nemo*.

When fuzzing starts, BELOBOG forks the Sui blockchain at checkpoint 148114817, right before the incident happens. Then, it initializes the on-chain object pool with available shared objects obtained from the forked chain. BELOBOG is capable of detecting the *SHL overflow* issue in a few seconds and reproducing the whole exploit with the *Earning Profits* oracle in less than 3 hours, where most of the time is spent on executing transactions.

**Nemo Incident Analysis.** The root cause of *Nemo* is an access control issue as presented in the Figure 14a. The function `calculate_amount_by_price()` is designed to estimate the output amount by the given amount and price. However, its underlying private function `current_price()` writes the user input price to its internal states. Therefore, anyone can potentially earn profits by calling the function to manipulate the price. Note that although triggering the vulnerability in Figure 14a is straightforward, it is not trivial to generate a full exploit because modifying the price does not yield profits directly. From a very high level, an attacker needs the following steps interacting with the functions, which is shown in Figure 14b. A successful exploit should: (1) borrow a flash loan via `borrow<T>()` to get some assets  $T_0$ , (2) call several `swap<T0, T1>()` and `get_oracle()` functions to swap the assets from  $T_0$  to  $T_1$ ; (3) manipulate the price of  $T_0$  and  $T_1$  with `calculate_amount_by_price()`; (4) swap  $T_1$  back to  $T_0$  with the shifted price; and (5) pay back the flashloan via `repay<T>()`. In practice, the actual exploit [8] composes a sequence of 231 Move function calls.

**Fuzzing Nemo with BELOBOG.** Thanks to the type graph, BELOBOG can always ensure the correct pairs of `swap<T0, T1>()` with `get_oracle()` and `borrow<T>()` with `repay<T>()`. This greatly reduces the sequence search space and enables BELOBOG to efficiently assemble the exploit with 10 Move function calls to trigger the *Earning Profits* oracle. BELOBOG forks the Sui blockchain at checkpoint 180415392, right before the *Nemo* attack, and fills in the related shared objects. It can generate a whole profitable exploit in less than 30 minutes.

## 6 Discussion

**Automated Testing Smart Contracts.** Both *Cetus* and *Nemo* studied in our evaluation are heavily audited by the leading Web3 auditing companies, but still, they fail to prevent these critical vulnerabilities. This suggests that humans are prone to making errors, and pure manual efforts are not enough to protect the security of smart contracts. Therefore, continuous and automated testing for the smart contracts is essential to secure deployed smart contracts. While there are already lots

of automated testing tools on the prevailing Ethereum Virtual Machine (EVM), we find that the Move ecosystem greatly lacks such tools and frameworks. Therefore, we hope that BELOBOG can shed light on the Move smart contracts testing by providing an extensible and general fuzz testing framework.

**Oracle Automation.** While BELOBOG has demonstrated its effectiveness in finding bugs and achieved high coverage and throughput, one of the limitations is that BELOBOG needs extra manual efforts to build oracles when testing real-world smart contracts that have a custom business model. We workaround this by providing a wide range of predefined oracles for common defects and allowing users to write oracles in Move language instead of modifying the fuzzer itself. It is an interesting direction to explore the automated fuzzing oracle synthesis for the Move smart contracts, given their strongly typed nature. It is possible to apply some existing approaches [17, 30–32] to extract high-level smart contract semantics to help build oracles automatically. We deem this as one of the interesting directions for our future research.

**State Dependency.** Although BELOBOG builds a type graph to model the type dependencies of various functions, it does not model the inner state dependency. In practice, smart contracts are stateful programs on the blockchain. Our evaluation suggests that our concolic executor could solve the direct dependency bound to the input parameters as the *Cetus* study in Section 5.5. It is also possible to integrate dataflow analysis to understand the state dependency, like Smartian[11]. We leave this as our future work.

## 7 Related Work

**Move Language Testing.** So far, there is a limited number of tools to test Move contracts automatically. Move Prover [34] pioneers in translating the Move language to its custom intermediate verification language to verify pre- and post-conditions. However, it requires manual effort to write specifications and only supports a limited set of oracles, making it less practical given the fancy business models of the DeFi ecosystem and learning costs [25]. MoveLint [2] is an open-sourced tool that detects several Move defects by analyzing the Abstract Syntax Tree of smart contracts. While it supports a wide range of oracles, it suffers from false positives due to the inaccuracy of static rules. MOVESCAN [25] instead translates the Move bytecode into stackless intermediate representation and generates Control Flow Graph to facilitate code analysis with extended oracle definitions. ItyFuzz [24] is originally designed for fuzzing bytecode on Ethereum Virtual Machine (EVM), it also ports its snapshot-based fuzzing approaches to MoveVM afterwards. SUI-FUZZER [3] is another coverage-guided fuzzer supporting both stateful and stateless fuzzing with custom properties.

**Smart Contract Fuzzing.** Fuzzing is an efficient way to test smart contracts. sFuzz [20] uses strategies adapted from AFL to improve the code coverage. Smartian [11] proposes both static and dynamic dataflow analysis to solve the complex data dependencies. ConFuzzius [27] hybrids symbolic execution, data dependency analysis, and taint analysis to exercise deeper bugs. VULS-EYE [18] novelly models the smart contract fuzzing as stateful directed fuzzing and guides the fuzzing by vulnerabilities. VERITE [16] builds a first profit-centric fuzzing framework with a gradient descent method to maximize the exploits. LLM4Fuzz [23] attempts to utilize the capability of Large Language Model (LLM) to guide the smart contract fuzzing. However, all of them focus on smart contracts on EVM and do not consider various type constraints in smart contracts. Thus, it is non-trivial to port them to support Move smart contracts as we have tried on ItyFuzz.

**API Fuzzing.** Generating a sequence of function calls to test the smart contract is similar to the traditional API fuzzing for libraries. GraphFuzz [13] models the execution trace as a dataflow graph to test the low-level library APIs. Minerva [35] utilizes dynamic mod-ref analysis to test

the browser APIs. APICraft [33] leverages both static and dynamic information to build fuzzer driver for testing closed-source SDKs. RESTler [10] as a stateful fuzzer analyzes the specification of the REST API by inferring the producer and consumer relations like our type graph building process. OGHARN [22] proposes the novel oracle-guided harnessing to automatically fuzz API. Due to the special requirements and constraints of smart contracts, applying such general API fuzzers to smart contracts cannot work as it is. However, it is interesting to explore how various API fuzzing insights can be applied in smart contract fuzzing.

## 8 Conclusion

In this paper, we present BELOBOG, the first effective fuzzing framework for Move smart contracts. We identified three key challenges in generating valid transactions for Move smart contracts, *i.e.*, type safety of objects, type parameters of functions, and type abilities of objects. In order to tackle these challenges, we propose a *type graph* guided fuzzing approach, where we use a type graph to model all the type constraints of the target smart contract. Upon fuzzing starts, BELOBOG will query the type graph to generate or mutate transactions that satisfy all the type requirements. To further tackle the strict checks that are common in smart contracts, we design and implement a concluder executor in BELOBOG.

## References

- [1] 2021. Move Lint. <https://www.soliditylang.org/>.
- [2] 2021. Move Lint. <https://github.com/BeosinBlockchainSecurity/Move-Lint>.
- [3] 2024. Sui-Fuzzer. <https://github.com/FuzzingLabs/sui-fuzzer>.
- [4] 2025. Cetus Incident Report: May 22, 2025 Attack Disclosure. <https://cetusprotocol.notion.site/Cetus-Incident-Report-May-22-2025-Attack-Disclosure-1ff1dbf3ac8680d7a98de6158597d416>.
- [5] 2025. DefiLlama. <https://apts.dev/build/smart-contracts/book>. Accessed on 13 Nov, 2025.
- [6] 2025. Move Lint. <https://github.com/foundry-rs/foundry>.
- [7] 2025. Move Subtyping. <https://move-language.github.io/move/tuples.html>.
- [8] 2025. The Nemo exploit transaction. <https://suivision.xyz/txblock/53B8psLGpomieU4dWkE4aP9txWHUnS7NWXmjmqPBWSBY>. Accessed on 13 Nov, 2025.
- [9] 2025. Nemo Security Incident: Cause, Process, and Fund Tracing Report V1.1. <https://olivine-hydrofoil-637.notion.site/Nemo-Security-Incident-Cause-Process-and-Fund-Tracing-Report-V1-1-26a6b8723d8a80e29cb8cb48fe1390f2>.
- [10] Vaggelis Attilidakis, Patrice Godefroid, and Marina Polishchuk. 2019. Restler: Stateful rest api fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 748–758.
- [11] Jaeseung Choi, Doyeon Kim, Soomin Kim, Gustavo Grieco, Alex Groce, and Sang Kil Cha. 2021. Smartian: Enhancing smart contract fuzzing with static and dynamic data-flow analyses. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 227–239. doi:10.1109/ASE51524.2021.9678888
- [12] Andrea Fioraldi, Dominik Christian Maier, Dongjia Zhang, and Davide Balzarotti. 2022. LibAFL: A Framework to Build Modular and Reusable Fuzzers. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles CA USA, 2022-11-07). ACM, 1051–1065. doi:10.1145/3548606.3560602
- [13] Harrison Green and Thanassis Avgerinos. 2022. Graphfuzz: Library api fuzzing with lifetime-aware dataflow graphs. In *Proceedings of the 44th International Conference on Software Engineering*. 1070–1081.
- [14] Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. 2020. Echidna: effective, usable, and fast fuzzing for smart contracts. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*. 557–560.
- [15] Timotej Kapus and Cristian Cadar. 2019. A segmented memory model for symbolic execution. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (Tallinn, Estonia) (ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 774–784. doi:10.1145/3338906.3338936
- [16] Ziqiao Kong, Cen Zhang, Maoyi Xie, Ming Hu, Yue Xue, Ye Liu, Haijun Wang, and Yang Liu. 2025. Smart Contract Fuzzing Towards Profitable Vulnerabilities. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 153–175.
- [17] Zihao Li, Jianfeng Li, Zheyuan He, Xiapu Luo, Ting Wang, Xiaozhe Ni, Wenwu Yang, Xi Chen, and Ting Chen. 2023. Demystifying DeFi MEV Activities in Flashbots Bundle. *Cryptology ePrint Archive*, Paper 2023/1855. doi:10.1145/3576915.3616590

- [18] Ruichao Liang, Jing Chen, Cong Wu, Kun He, Yueming Wu, Ruochen Cao, Ruiying Du, Ziming Zhao, and Yang Liu. 2025. Vulseye: Detect smart contract vulnerabilities via stateful directed graybox fuzzing. *IEEE Transactions on Information Forensics and Security* (2025).
- [19] David Mandelin, Lin Xu, Rastislav Bodik, and Doug Kimelman. 2005. Jungloid mining: helping to navigate the API jungle. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) (PLDI '05). Association for Computing Machinery, New York, NY, USA, 48–61. doi:10.1145/1065010.1065018
- [20] Tai D Nguyen, Long H Pham, Jun Sun, Yun Lin, and Quang Tran Minh. 2020. sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 778–788. doi:10.1145/3377811.3380334
- [21] Kaihua Qin, Liyi Zhou, Benjamin Livshits, and Arthur Gervais. 2021. Attacking the defi ecosystem with flash loans for fun and profit. In *International conference on financial cryptography and data security*. Springer, 3–32.
- [22] Gabriel Sherman and Stefan Nagy. 2025. No Harness, No Problem: Oracle-guided Harnessing for Auto-generating C API Fuzzing Harnesses. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 775–775.
- [23] Chaofan Shou, Jing Liu, Doudou Lu, and Koushik Sen. 2024. Llm4fuzz: Guided fuzzing of smart contracts with large language models. *arXiv preprint arXiv:2401.11108* (2024).
- [24] Chaofan Shou, Shangyin Tan, and Koushik Sen. 2023. ItyFuzz: Snapshot-Based Fuzzer for Smart Contract. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle WA USA, 2023-07-12). ACM, 322–333. doi:10.1145/3597926.3598059
- [25] Shuwei Song, Jiachi Chen, Ting Chen, Xiapu Luo, Teng Li, Wenwu Yang, Leqing Wang, Weijie Zhang, Feng Luo, Zheyuan He, Yi Lu, and Pan Li. 2024. Empirical Study of Move Smart Contract Security: Introducing MoveScan for Enhanced Analysis. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna Austria, 2024-09-11). ACM, 1682–1694. doi:10.1145/3650212.3680391
- [26] Thodoris Sotiropoulos, Stefanos Chaliasos, and Zhendong Su. 2024. API-Driven Program Synthesis for Testing Static Typing Implementations. *Proc. ACM Program. Lang.* 8, POPL, Article 62 (Jan. 2024), 32 pages. doi:10.1145/3632904
- [27] Christof Ferreira Torres, Antonio Ken Iannillo, Arthur Gervais, and Radu State. 2021. Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 103–119. doi:10.1109/EuroSP51992.2021.00018
- [28] Adam Welc and Sam Blackshear. 2023. Sui move: Modern blockchain programming with objects. In *Companion Proceedings of the 2023 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*. 53–55.
- [29] Dr Gavin Wood. 2024. ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER SHANGHAI VERSION. <https://ethereum.github.io/yellowpaper/paper.pdf>. (2024).
- [30] Cong Wu, Jing Chen, Ziwei Wang, Ruichao Liang, and Ruiying Du. 2024. Semantic sleuth: Identifying ponzi contracts via large language models. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 582–593.
- [31] Siwei Wu, Zhou Yu, Dabao Wang, Yajin Zhou, Lei Wu, Haoyu Wang, and Xingliang Yuan. 2023. Defiranger: Detecting defi price manipulation attacks. *IEEE Transactions on Dependable and Secure Computing* 21, 4 (2023), 4147–4161.
- [32] Maoyi Xie, Ming Hu, Ziqiao Kong, Cen Zhang, Yebo Feng, Haijun Wang, Yue Xue, Hao Zhang, Ye Liu, and Yang Liu. 2024. DeFort: Automatic Detection and Analysis of Price Manipulation Attacks in DeFi Applications. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) (ISSTA 2024). Association for Computing Machinery, New York, NY, USA, 402–414. doi:10.1145/3650212.3652137
- [33] Cen Zhang, Xingwei Lin, Yuekang Li, Yinxing Xue, Jundong Xie, Hongxu Chen, Xinlei Ying, Jiashui Wang, and Yang Liu. 2021. {APICraft}: Fuzz driver generation for closed-source {SDK} libraries. In *30th USENIX Security Symposium (USENIX Security 21)*. 2811–2828.
- [34] Jingyi Emma Zhong, Kevin Cheang, Shaz Qadeer, Wolfgang Grieskamp, Sam Blackshear, Junkil Park, Yoni Zohar, Clark Barrett, and David L. Dill. 2020. The Move Prover. In *Computer Aided Verification*, Shuvendu K. Lahiri and Chao Wang (Eds.). Vol. 12224. Springer International Publishing, 137–150. doi:10.1007/978-3-030-53288-8\_7 Series Title: Lecture Notes in Computer Science.
- [35] Chijin Zhou, Quan Zhang, Mingzhe Wang, Lihua Guo, Jie Liang, Zhe Liu, Mathias Payer, and Yu Jiang. 2022. Minerva: browser API fuzzing with dynamic mod-ref analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1135–1147.
- [36] Liyi Zhou, Xihan Xiong, Jens Ernstberger, Stefanos Chaliasos, Zhipeng Wang, Ye Wang, Kaihua Qin, Roger Wattenhofer, Dawn Song, and Arthur Gervais. 2023. Sok: Decentralized finance (defi) attacks. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2444–2461. doi:10.1145/3558535.3559780