# Towards Smart Contract Fuzzing on GPUs

Weimin Chen[†], Xiapu Luo[†✉], Haipeng Cai[‡], Haoyu Wang[§]
[†]The Hong Kong Polytechnic University, China
[‡]Washington State University, United States
[§]Huazhong University of Science and Technology, China
[†]{cswchen, csxluo}@comp.polyu.edu.hk, [‡]haipeng.cai@wsu.edu, [§]haoyuwang@hust.edu.cn

*Abstract*—**Fuzzing is one of the major techniques for uncovering vulnerabilities in smart contracts. The effectiveness of fuzzing is significantly affected by its throughput but unfortunately existing fuzzers for smart contracts have low throughput due to the slow execution of EVM, the delay introduced by the consensus protocols, the limited parallelization capability of CPUs, and the overhead caused by the instrumented EVM. To tackle this critical issue, in this paper, we take the first step to leverage GPU's parallel computing power to boost the throughput of smart contract fuzzing. More precisely, by converting the fuzzing workload to a SIMD task, we can activate thousands of GPU cores to test the smart contract simultaneously. To achieve this purpose, we design new solutions to address three major challenges, namely developing incremental storage to reduce GPU memory cost, proposing a stateful bitmap to embed transaction dependency to the feedback metric, and designing a parallel feedback algorithm to rule out undesired seeds that cause redundant overlaps. We implement a prototype named MAU, which first transforms the bytecode of a smart contract to a SIMD application in PTX assembly and then runs it parallelly on the GPU. We evaluate MAU using both a large and small benchmark. The experimental results demonstrate that the throughput of MAU reaches 162.37K execs/sec and 328.06K execs/sec, which leads to an 8.69-15.38X improvement to the state-of-the-art tool. Moreover, the high throughput empowers MAU to detect 1.01-2.50X more bugs and obtain 1.03–4.71X more code coverage than baselines.**

## 1. Introduction

Smart contracts are decentralized applications that manage cryptocurrency, i.e., Ether and ERC tokens. The prosperity of blockchain ecosystems relies on various applications (e.g., DeFi) based on smart contracts. Unfortunately, recent years have witnessed many attacks that exploit the vulnerabilities in smart contracts and lead to billions of financial losses [59], [33], [20], [43], [53], [30], [26], [36], [9], [38], [31]. Therefore, many approaches have been proposed to detect vulnerabilities in smart contracts [21], [51], [65]. Among them, fuzzing has been widely used to successfully uncover many vulnerabilities in smart contracts [32], [47], [27], [11], [66], [25], [62], [10], [58]. We focus on Ethereum smart contracts because Ethereum blockchain supports the largest number of smart contracts with over one million daily transactions [29]. Six of the top ten popular blockchains by market capitalization [12] are compatible with Ethereum.

✉ *The corresponding author.*

TABLE 1: Comparison of existing smart contract fuzzers. OC: open-source. Bytecode: the tool can run without the source code of the target smart contract. Throughput: the number of seeds evaluated by the tool within one second. Novelty: the main contribution aspects. We did not test Harvey because it is closed-source.

| Tool | OC | Bytecode | Throughput | Novelty |
|---|---|---|---|---|
| Ityfuzz [54] | ✓ | ✓ | 33.0K | ②③ |
| Smartian [11] | ✓ | ✓ | 419 | ③ |
| EF\CF [52] | ✓ | ✓ | 23.9K | ② |
| sFuzz [47] | ✓ | ✓ | 512 | ③ |
| ILF [27] | ✓ | ✓ | 57 | ① |
| Echidna [25] | ✓ | ✗ | 166 | ③ |
| Harvey [66] | ✗ | ✗ | - | ①③ |
| ConFuzzius [62] | ✓ | ✗ | 107 | ③ |

Fuzzing typically aims to generate test cases that are expected to test as many execution states of the target program as possible. Since smart contracts are stateful [11], the fuzzer usually needs to generate a sequence of transactions as the test case (a.k.a. seed). To be specific, smart contract fuzzing is an iteration process consisting of three typical steps in each round, such as ① mutating a seed to derive a new one; ② executing the smart contract with the mutated seed; ③ gathering feedback metrics to guide the next round, such as code coverage [11], branch distance [47], machine learning [27], and transaction dependency [11], [62], [54]. The seed with a higher value (known as interesting) has more chances to be selected in the following rounds.

The effectiveness of fuzzing is significantly affected by throughput [7], i.e., the number of seeds evaluated per second. A linear increase in code coverage approximately requires exponentially more CPU resources [6]. To uncover potential opportunities for improving throughput, we conducted a motivating study to profile the distribution of time spent by fuzzing tools across Steps ①-③ (§ 2.4). While researchers have invested tremendous efforts on improving Steps ① and Step ③ with smarter mutation strategies [32], [47], [27] and prioritizing valuable seeds [54], [47], [25], [62], [11], limited research [52] focuses on improving Step ② (Execution), which spent over 70% of testing time. Table 1 lists the throughput of eight state-of-the-art fuzzers. The throughput data comes from [65], which runs the tools on a small benchmark with an average number of 224 lines of code. Although researchers have proposed various optimizations for fuzzing, time-consuming factors are still hindering the throughput, thereby preventing smart contract fuzzers from obtaining higher code coverage and better performance. Specifically, these fuzzers execute the smart contracts in EVM (Ethereum Virtual Machine) with the underlying consensus protocols in a single CPU thread and interrupt EVM to collect feedback metrics. Unfortunately, their throughput is significantly restricted by the slow execution of EVM, the delay introduced by the underlying consensus protocols, the limited parallelization capability of CPUs, and the overhead caused by the EVM instrumentation. EF\CF is the only tool optimized for ②, however, it is even slower than the conventional fuzzer like Ityfuzz. Smartian [11] is the third follower. It achieves only 419 execs/sec, even though it has used an efficient EVM interpreter, i.e., Nethermind.

Task parallelism is a straightforward yet costly approach to improve throughput. However, fuzzers using large-scale machines or clusters still confront throughput limits, as noted in [22], [39], [57]. This issue is further compounded when fuzzing smart contracts, which run on slower interpreter-based code rather than native code. Yet it has been shown that linearly more compute resources only leads to finding the same known bugs linearly faster while finding linearly more (new) bugs requires exponentially more compute resources [6]. Thus, to significantly accelerate new vulnerability discovery, increasing fuzzing efficiency by *orders of magnitude* is needed, for which leveraging GPU is promising. For instance, one NVIDIA RTX3090 GPU has 10,496 cores, whereas a typical CPU has at most 128 cores.

In this paper, we design and develop MAU, the *first* GPU-based fuzzer for smart contracts to improve throughput exponentially by leveraging GPU's parallel computing power and exploiting the features of smart contract fuzzing. More precisely, although GPUs are not designed for general computation, we can convert the smart contract fuzzing to a GPU-friendly application, i.e., SIMD code (Single Instruction, Multiple Data [61]) and then discard the EVM interpreter. Moreover, we observe four properties of smart contract fuzzing that make it possible to leverage GPU to boost the throughput: **P1)** No heavy synchronization among threads is required because each fuzzing round is relatively independent. That is, smart contracts are usually free of locks; **P2)** The SIMD overhead due to the branch divergence [13] could be eased because most seeds are uninteresting and execute duplicate paths. Note that preemptively omitting all these uninteresting seeds is impractical, as their redundancy only becomes apparent post-evaluation. **P3)** The execution of Ethereum smart contracts is serial; **P4)** Smart contracts do not utilize system calls like Libc. Due to P3 and P4, we can convert the target smart contract to a standalone application on the GPU.

Given the bytecode of a smart contract, MAU first transforms/rewrites it into a PTX assembly [49] that can run on a GPU through a functional equivalent LLVM IR [37]. The instruction lifting (§ 3.2) converts EVM instructions to LLVM assembly, which are further extended with code vectorization (§ 3.3) for a SIMD-based fuzzing. The rewritten IR is translated to a PTX assembly, which MAU executes in a GPU to evaluate the seeds in terms of code coverage and transaction dependency (§ 3.4).

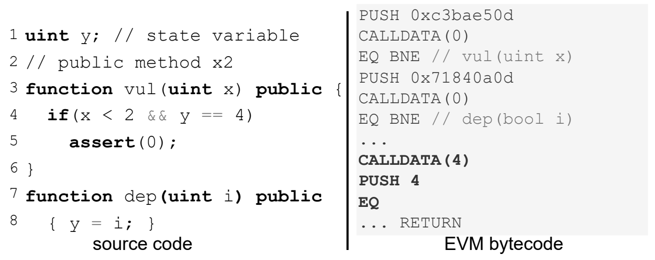
It is non-trivial to develop MAU due to three challenges. **C1:** Storage is essential for smart contract execution. However, migrating the EVM storage to GPU is expensive because the EVM specification defines that each storage should have a size of $2^{256} * 32$ bytes, which far exceeds the GPU's capability. **C2:** During smart contract execution, tracking transaction dependency in an instrumented EVM (CPU side) will introduce a non-negligible overhead to fuzzing. **C3:** Based on our observation, only a few seeds can find new paths. While seeds evaluated together on the GPU can improve throughput, potential overlapped execution [40] may cause redundant seeds and prevent the fuzzer from reaching a higher throughput.

To tackle **C1**, we use incremental snapshot design to simulate the storage on GPU (§ 4.2.2). It can sharply reduce redundant GPU memory cost. To address **C2**, we propose a stateful bitmap in § 4.3.2 to gather transaction dependency efficiently. By instrumenting the smart contract rather than the EVM interpreter, the stateful bitmap measures the transaction sequence in terms of the data dependency between the state variables. To approach **C3**, we design a parallel feedback algorithm to rule out undesired seeds that cause redundant overlaps (§ 4.3.3).

We implemented the prototype of MAU with around 8,000 lines of C++ and 100 lines of Rust. We carefully evaluated it using a large benchmark with an average of 13K LoC and a small benchmark with ground truth. The experimental results demonstrate that the throughput of MAU reaches 162.37K execs/sec and 328.06K execs/sec in the large dataset and small dataset, respectively. It leads to an 8.69X-15.38X improvement to the fastest baseline. Moreover, the high throughput empowers MAU to achieve 3.36%-371.43% more code coverage and detect 0.71%-150.32% more bugs than baselines.

**Our contributions:**

- We design and develop MAU, the first GPU-based fuzzer for smart contracts, by leveraging GPU's parallel computing power and exploiting the features of smart contract fuzzing.
- We propose new solutions to tackle three challenging issues in the development of MAU and implement the prototype, which is available at Figshare.
- We evaluate MAU using comprehensive benchmarks in terms of various metrics. The experimental results show that MAU can detect 1.01-2.50X more bugs and obtain 1.03-4.71X more code coverage than baselines, with an 8.69-15.38X throughput improvement (§ 5.3).



```
1 uint y; // state variable
2 // public method x2
3 function vul(uint x) public {
4   if(x < 2 && y == 4)
5     assert(0);
6 }
7 function dep(uint i) public
8   { y = i; }
```

```
PUSH 0xc3bae50d
CALLDATA(0)
EQ BNE // vul(uint x)
PUSH 0x71840a0d
CALLDATA(0)
EQ BNE // dep(bool i)
...
CALLDATA(4)
PUSH 4
EQ
... RETURN
```

source code
EVM bytecode

Figure 1: The EVM bytecode (right side) is compiled from the source code in Solidity (left side).

- We perform a systematic evaluation of MAU on 183,111 real-world smart contracts. MAU identified vulnerabilities in 548 projects, holding assets exceeding \$172K in value (§ 5.5). The detailed statistics about the found bugs are included in the artifact[1].

## 2. Background

This section provides an overview of SIMD and smart contract fuzzing, followed by a motivating study that explores opportunities for enhancement.

### 2.1. SIMD via CUDA

SIMD (Single Instruction, Multiple Data) is a parallel architecture in which a single instruction controls multiple processing units. In the realm of GPUs, this parallelism is typically implemented through CUDA [48]. While all threads start from the same instruction address, they execute independently within their individual contexts, including registers and stacks. Data parallelism [60] focuses on distributing the data across different threads, which solve subtasks on the data in parallel. PTX [49] is the GPU instruction set tailored for SIMD. Any exception thrown from a PTX application is sticky, resulting in a corrupted GPU context. In practice, the occurrence of a sticky error terminates all other running threads, even if the error is isolated to one particular thread. Therefore, it is important to make the PTX program gently exit from the GPU environment when handling the exceptions defined in the EVM specification.

### 2.2. Smart Contract

Ethereum smart contracts are typically written in Solidity [18], including a set of state variables declared globally and a series of methods designed to manipulate these variables. Smart contract source code is compiled into EVM bytecode, which is a sequence of instructions. The EVM, acting as the execution environment, interprets EVM bytecode with the given input (i.e., transaction). During transaction execution, EVM instructions: 1) load input arguments from *calldata*; 2) manage runtime data in the *stack*; 3) store temporary data (i.e., local variables) in the *memory*; 4) and store persistent data (i.e., state variables) in the *storage*. In particular, *calldata* is an input buffer, maintaining the invoked function's signature and arguments.

1. https://figshare.com/s/2fd3f1616a17e78ffee7

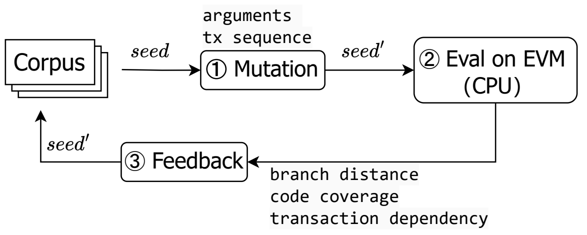


Figure 2: A typical workflow of smart contract fuzzing.

Smart contracts are stateful, with persistent state variables on the blockchain. These variables can affect the execution path. Specifically, state variables often participate in branch conditions, restricting access to certain code segments until the dataflow dependency of state variables has been satisfied by a predefined transaction. We call this hurdle transaction dependency. As shown in the example in Figure 1, the bug at line 5 can only be triggered when the conditional branch at L4 satisfies expected values for `x` and `y`. To hit the bug, we have to first set the state variable $y$ to 4 by executing `dep(4)`, then explore the transaction argument $x$ to meet the condition at L4. Resolving transaction dependency remains an open question [11], [56]. It is challenging to identify expected transaction sequences due to the implicit dataflow dependency in smart contracts, especially for binary-only testing.

### 2.3. Smart Contract Fuzzing

Smart contract fuzzing is a technique aimed at searching test cases (i.e., seeds) that a target smart contract can exercise to explore as many program states as possible. It is an iterative process consisting of three typical steps, as outlined in Figure 2. During each iteration, the fuzzer ① mutates a seed from the corpus to derive a new one; ② executes the target smart contract in EVM to evaluate the mutated seed; ③ gathers feedback metrics like code coverage and transaction dependency to assess the seed's value. If a seed has enough values, the fuzzer will take it as interesting and add it to the corpus for the next fuzzing round.

While many studies have focused on improving Step ① and ③ with smarter mutation strategies [32], [47], [27] and prioritizing valuable seeds [47], [25], [66], this paper concentrates on improving Step ② (Execution), which few work has been devoted to. It is motivated by our observation that fuzzers are more effective when the throughput is improved, e.g., fuzzing on a faster CPU with SSD [69]. Throughput is defined as the number of seeds evaluated by the fuzzer per second. Specifically, we aim to examine and rectify design elements in fuzzing tools that may hinder iteration speed, thereby enhancing overall throughput and fuzzing performance.

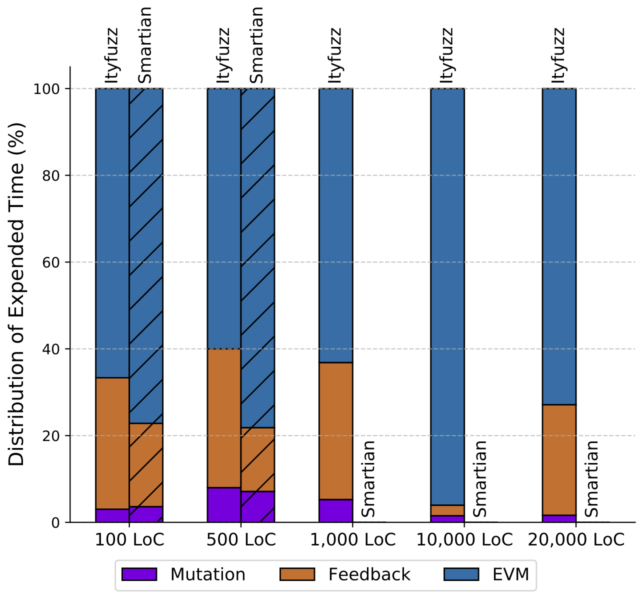


Figure 3: Distribution of time spent by Ityfuzz and Smartian across three typical steps in one fuzzing iteration. In the legend, Mutation, EVM, and Feedback represent steps ①, ②, and ③, respectively.

## 2.4. Motivating Study

To uncover potential opportunities for improving fuzzing performance, we conduct a motivating study to profile the distribution of time spent by fuzzing tools across the three typical steps in one fuzzing iteration.

**Experimental Environment.** We randomly selected five real-world smart contracts with varying source code sizes, ranging from 100 LoC to 20,000 LoC. As for fuzzers, we customized Ityfuzz [54] and Smartian [11] to examine time distribution at each fuzzing step. Ityfuzz was chosen because it is the only fuzzer based on LibAFL, an efficient framework adopted by many other tools [68], [42]. Smartian was also included because it is a conventional fuzzer optimized for resolving transaction dependency. Although EF\CF [52] is the only tool focusing on improving Step ②, Ityfuzz can outperform it in terms of throughput. This is also the reason why we pick Ityfuzz rather than EF\CF. We did not select more fuzzers because Ityfuzz and Smartian have already been shown to significantly outperform other baselines in their papers. We ran both tools for ten minutes on the five smart contracts. All experiments were conducted in the same environment we used at § 5.

**Results.** Overall, the primary bottleneck of the throughput is the slow execution within EVM. Figure 3 visualizes the average time distribution across step ①-③. Smartian crashed when testing the three complex examples larger than 1,000 LoC, resulting in missing performance data. For Ityfuzz and Smartian, 83% and 78% of the total time was spent on Step ②, highlighting EVM execution as a significant time-consuming factor. In contrast, Ityfuzz allocated 2% and 15% of total time to Step ① and Step ③. Smartian spent 6% and 16% of the time running Step ① and Step ③, respectively. Regarding smart contract size, the study generally revealed a correlation: the more complex the smart contract is, the more time spent on EVM execution. This observation suggests that there are potential opportunities to improve throughput by optimizing EVM execution, particularly when testing complex blockchain projects.

**Main Idea.** To improve the fuzzing throughput, our intuitive solution is to boost Step ② by concurrently evaluating multiple seeds on a SIMD GPU. More specially, MAU translates the smart contract to the GPU machine code, i.e., PTX assembly, and then evaluates multiple seeds in a SIMD workflow. GPU is cheaper than CPU in solving parallel tasks. In Google Cloud, 1,024 GPU cores on a single Tesla T4 ($0.11/h) are cheaper than one CPU core ($0.14/h). We can allocate more cores to MAU, potentially yielding further performance improvements (see § 5.3).

Smart contract fuzzing is GPU-friendly due to the following four aspects. Fuzzing threads execute smart contracts independently with assigned seeds, making the smart contract free of locks. This means **P1**: the heavy synchronization mechanism is unnecessary. Another common feature of fuzzing is **P2**: the redundant execution eases the branch divergence overhead[2]. During fuzzing, only a few seeds can eventually hit new paths. Although most seeds are redundant, it is impossible to skip them because we do not know a seed is redundant until it has been evaluated. GPU can boost the fuzzing because most seed evaluations involve similar paths. The other two properties are about smart contract primitives. **P3:** the execution of smart contracts is serial, aligning well with GPU-accelerated fuzzing; **P4:** system calls are not required for executing smart contracts. Due to P3 and P4, the target smart contract can be converted into a standalone SIMD application on the GPU, making GPU-accelerated fuzzing an ideal fit for smart contract fuzzing.

For a preliminary analysis, we compare the state-of-the-art tool Ityfuzz [54] (utilizing one CPU core) against MAU (leveraging one CPU core and 256 GPU cores) to analyze the illustrated puzzle in Figure 1. Ityfuzz requires 13 minutes to execute 23 million fuzzing iterations before identifying the buggy code, while MAU can accomplish the same task within 6.5 minutes or 20 million iterations, demonstrating a 2X speed improvement.

## 3. Overview

In this section, we briefly introduce the architecture of MAU and its solutions to the three technical challenges.

### 3.1. MAU's Architecture

Figure 4 depicts the architecture of MAU, which takes EVM bytecode as input and outputs a set of seeds to explore

2. Effect of executing a branch where, for some threads, the branch is taken, and for other(s), it is not taken. These two groups of threads will then run in serial rather than parallel.

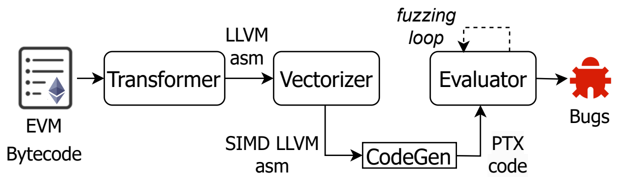


Figure 4: The overview of MAU.

as many program states of the smart contract as possible. MAU consists of three main components: (1) Transformer, (2) Vectorizer, and (3) Evaluator. Transformer lifts the EVM bytecode to an LLVM assembly, a register-based representation. Vectorizer applies code vectorization to the LLVM assembly, generating a SIMD program that tracks smart contract execution states in the GPU environment regarding code coverage and transaction dependency. During the CodeGen phase, the LLVM backend translates the vectorized assembly into PTX code, which can run on the GPU. Evaluator schedules and evaluates a batch of seeds in parallel to improve fuzzing throughput. It employs a CPU-GPU task decomposition strategy to generate seeds that deserve more testing. Whenever the GPU-evaluated seeds explore new program states of the smart contract, Evaluator validates them on the CPU for correctness. Therefore, GPU-accelerated fuzzing virtually produces no false positives because the seeds are eventually executed in the EVM to confirm the problem. If a bug is detected before the timeout, Evaluator outputs its bug location (i.e., instruction address). Throughout the entire fuzzing, Transformer only needs to run once for PTX generation. It is worth noting that MAU does not require the source code of the smart contract.

## 3.2. Transformer

Given the EVM bytecode, Transformer lifts it to a register-based representation in LLVM IR while remaining EVM functionalities. We use intermediate representation as the link between EVM bytecode and PTX code because the register-based IR offers a uniform platform for code vectorization and instrumentation.

Transformer uses LLVM memory to represent the components used by EVM (i.e., stack, memory, calldata, and storage). The stack-based EVM instructions are devirtulated to register-based LLVM memory operations (see § 4.1). For the stack, Transformer employs a byte vector (i.e., $\mu$) along with an additional register $p$. $p$ maintains the current stack depth, so $\mu[p]$ is always the item in the stack top. Since EVM instructions are stack-based, Transformer can obtain the instructions operands from $\mu$ by adjusting the value of $p$. For instance, the EVM instruction `ADD` is expected to 1) pop its two operands from the stack and push back the result. It can be lifted to 1) $c \leftarrow \mu[p] + \mu[p-1]; p \leftarrow p-2$; 2) $p \leftarrow p+1; \mu[p] \leftarrow c$. Additionally, EVM memory is a byte-addressing vector; thus, Transformer uses a global byte vector, named $v$, to represent it. calldata is the input buffer of the smart contract; thus, Transformer declares a byte vector (i.e., $d$) to represent it. Especially, it is expensive to use memory to simulate storage in the GPU environment because the storage of each smart contract at most has $2^{256}$ slots by design (**C1**). EVM can read and write any slot with a 32-byte address. To avoid redundant memory cost, in § 4.2.2, we use an incremental snapshot design as the storage, denoted as $\sigma$.

## 3.3. Vectorizer

Taking the LLVM assembly as the input, Vectorizer aims to generate a SIMD program in LLVM IR. This SIMD code is expected to evaluate a batch of seeds together to improve the fuzzing throughput. To this end, Transformer first applies code vectorization to rewrite the LLVM IR (§ 4.2) and then instruments a SIMD-friendly bitmap to track the execution states (§ 4.3.2).

Vectorizer converts the LLVM instructions to vector operations using data parallelism [16]. The LLVM memory used in Transformer (i.e., $\mu$, $v$, $d$, and $\sigma$) are extended to thread-safe vectors, i.e., $\vec{\mu}$, $\vec{v}$, $\vec{d}$, and $\vec{\sigma}$. Each SIMD thread can access its thread-local data based on its thread ID, denoted as $i$. For example, every SIMD thread can execute the same PTX code, e.g., `load` $\vec{d}_{\mathtt{i}}[0...3]$ to load the first four bytes of from their individual calldata and evaluate them together.

In addition, Vectorizer applies instrumentation to smart contracts instead of EVM in order to track the seed execution states in the GPU environment (§ 4.3). An interesting seed contributes to exploring the program space in terms of both code coverage and transaction dependency. We adopt code coverage because the seeds that hit new code are more likely to explore deeper program space [45]. Since smart contracts are stateful, the seed should also be interesting if it can drive the smart contract to an expected state. We measure the transaction dependency in terms of the dataflow dependency (read-after-write) between the state variables. If more dataflow dependencies of state variables are found, we believe the evaluated seed contains a transaction sequence that deserves more tests.

Existing tools [11], [62] track read-after-write dataflow in an instrumented EVM, which introduces non-negligible overhead (**C2**). To overcome this challenge, we design a stateful bitmap in § 4.3.2 to gather transaction dependency efficiently.

## 3.4. Evaluator

The established LLVM assembly can be naturally translated into PTX assembly by the LLVM backend [41]. Evaluator executes the PTX assembly on a GPU to search for interesting seeds and output any possible bugs as defined by the oracles. Figure 5 is the overview of Evaluator, which applies a CPU-GPU task decomposition strategy to generate seeds that deserve more testing. In every fuzzing round, Evaluator utilizes GPU mutation (GPM) to select one seed from the corpus and mutates it to generate multiple variants in GPU memory (§ 4.5). These threads will run in the

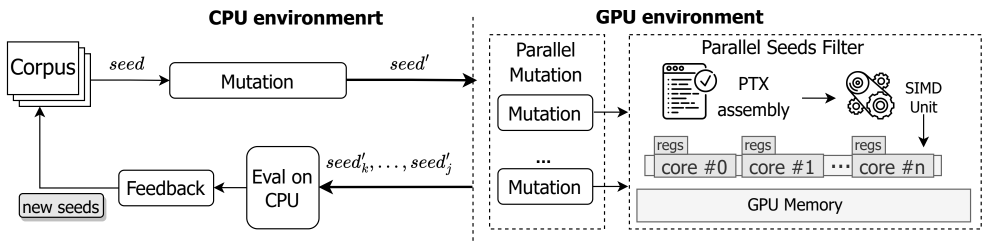


Figure 5: The overview of Evaluator.

GPU environment in parallel to boost the seed evaluation. We need GPM because it can avoid frequent data transfers between the CPU and GPU, which would otherwise reduce throughput. Additionally, we validate the interesting seeds from the GPU in the CPU to avoid false positives due to the potential incorrect code translation if any. Bugs are detected by the oracles (§ 4.4). In the end, the interesting seeds will be eventually added to the corpus.

The seeds evaluated together on the GPU may hit the same paths. The potential overlapped execution [40] may prevent the multi-thread fuzzer from achieving higher throughput (**C3**). To tackle this problem, we design a parallel feedback algorithm (SBF) to rule out undesired seeds that cause redundant overlaps (§ 4.3.3).

## 4. Mau Design

In this section, we present the technical details of Mau.

### 4.1. Lifting EVM Instructions

To be specific, the processing pipeline consists of four steps. First, Transformer constructs a CFG from the EVM bytecode (§ 4.1.1). Second, it uses LLVM memory operations to emulate the EVM instructions based on the CFG (§ 4.1.2). Third, it recovers control flow in the generated LLVM IR by resolving jumps (§ 4.1.3). At last, it recovers EVM exception-handling in the GPU environment (§ 4.1.4).

**4.1.1. Constructing Basic Blocks.** Given the EVM bytecode, we identify EVM basic blocks and then generate the same number of basic blocks in an on-the-fly LLVM function. EVM specification defines basic blocks according to two rules: 1) a basic block must end with a terminator (i.e., `JUMP`, `JUMPI`, `RETURN`, `STOP`, `REVERT`, and `SUICIDE`); 2) `JUMPDEST` is the jump destination. These two rules enable us to identify all basic blocks using a linear sweep. As a result, all possible destinations of the indirect jumps used in EVM bytecode can be over-approximated. This is critical for resolving indirect jumps in § 4.1.3. For example, all indirect destinations are recovered.

**4.1.2. Generating IR.** In each LLVM basic block, we generate LLVM instructions from the EVM instructions. We use arrays in LLVM memory to emulate EVM components (i.e., *stack*, *memory*, *storage*, and *calldata*). In addition,

TABLE 2: The emulation rules of the code translation. $\mu$, $v$, $d$, and $\sigma$ represent the LLVM arrays for simulating the EVM stack, memory, calldata, and storage, respectively. $p$ is a local register maintaining the current stack depth. `switch` is a table jump in LLVM IR used to lift the EVM indirect jumps. Its jump table contains all destinations.

| EVM Instruction | Translation Rules |
|---|---|
| `push` $c$ | $p \leftarrow p+1; \mu[p] \leftarrow c$ |
| `pop` | $p \leftarrow p-1$ |
| `dup` $c$ | $p \leftarrow p+1; \mu[p] \leftarrow \mu[p-c]$ |
| `swap` $c$ | $t \leftarrow \mu[p];$ $\mu[p] \leftarrow \mu[p-c]; \mu[p-c] \leftarrow t$ |
| `mload` | $\mu[p] \leftarrow v[\mu[p]...(\mu[p]+31)]$ |
| `mstore` | $v[\mu[p]...(\mu[p]+31)] \leftarrow \mu[p-1];$ $p \leftarrow p-2$ |
| `sload` | $\mu[p] \leftarrow \sigma[\mu[p]]$ |
| `sstore` | $\sigma[\mu[p]] \leftarrow \mu[p-1]; p \leftarrow p-2$ |
| `calldata` | $x, y, z \leftarrow \mu[p], \mu[p-1], \mu[p-2];$ $v[x...(x+z)] \leftarrow d[y...(y+z)]$ |
| unary `op` | $\mu[p] \leftarrow$ `op` $\mu[p]$ |
| binary `op` | $c \leftarrow \mu[p]$ `op` $\mu[p-1]; p \leftarrow p-2;$ $p \leftarrow p+1; \mu[p] \leftarrow c$ |
| `sha3` | $x, y \leftarrow \mu[p], \mu[p-1]; p \leftarrow p-1;$ $\mu[p] \leftarrow keccak(v[x...(x+y)])$ |
| `call` | $p \leftarrow p-4; \mu[p] \leftarrow true$ |
| `getenv` $env$ | $p \leftarrow p+1; \mu[p] \leftarrow env$ |
| `bne` | $t, c \leftarrow \mu[p], \mu[p-1]; p \leftarrow p-2;$ br $c \neq 0, t,$ `<fall_through>`; |
| `br` | $t \leftarrow \mu[p]; p \leftarrow p-1;$ `switch` $t$, `<destinations>`; |
| `stop` | `ret true` |
| `revert` | `ret false` |
| `suicide` | move the self balance to $\mu[p]$; `ret true` |

determining the operands of EVM instructions from the EVM bytecode is non-trivial due to the implicit use-define chain [23], [24], [34]. Specifically, some EVM instructions may use data defined by the other instructions in prior basic blocks, resulting in an implicit use-define chain because of the indirect jumps between EVM basic blocks. To solve this problem, we use LLVM memory operations to emulate the EVM instructions. The operands of EVM instructions are emulated by LLVM memory data without relying on the use-define chain. To better explain the emulation approach, we next describe the EVM instruction set with a formal model and show the translation rules.

$$\begin{aligned} bytecode = &\ ins \mid bytecode \\ ins = &\ \texttt{push}\ c \mid \texttt{pop} \mid \texttt{swap} \mid \texttt{dup} \mid \texttt{op} \mid \texttt{sha3} \mid \texttt{calldata} \mid \\ &\ \texttt{mload} \mid \texttt{mstore} \mid \texttt{sload} \mid \texttt{sstore} \mid \texttt{call} \mid \texttt{getenv} \mid \\ &\ \texttt{br} \mid \texttt{bne} \mid \texttt{suicide} \mid \texttt{stop} \mid \texttt{revert} \end{aligned}$$

The EVM instruction is a stack-based language. Besides executing general calculations (`op` and `sha3`) and execution control (`br`, `bne`, `suicide`, `stop`, and `revert`), it can use data from various EVM components, such as stack (`push`, `pop`, `swap`, and `dup`), memory (`mload` and `mstore`), calldata (`calldata`), and storage (`sload` and `sstore`). Moreover, EVM can interact with blockchain through `call` and `getenv`. Table 2 shows the emulation rules for EVM instructions. In particular, the EVM exceptions are handled in § 4.1.4.

**stack instruction.** EVM can operate its stack by `push`, `pop`, `swap`, and `dup`. We build a simulated stack for the EVM stack instructions, which includes an LLVM array $\mu$ to store data and a register $p$ to track the stack depth. $\mu[p]$ indicates the item at the top of the EVM stack. As a result, we can determine all operands of EVM instructions from $\mu$. The EVM instruction `push` $c$ is expected to push the constant $c$ onto the EVM stack. We can lift it to $p \leftarrow p + 1; \mu[p] \leftarrow c$. Namely, `pop` is used to pop the top item from the EVM stack, which we lift to $p \leftarrow p - 1$. In particular, `swap c` exchanges the 1st and $c$-th stack items, and `dup c` duplicates the $c$-th stack item. The translation rules for these two kinds of instructions are shown in Table 2.

**memory instruction.** EVM has a linear memory, with `mload` and `mstore` accessing data in bytes. We use an LLVM array $v$ as the EVM memory. `mload` and `mstore` are emulated by LLVM memory operations. Note that the data from the emulated memory is converted to little-endianness because GPU is a little-endian architecture, but EVM data is big-endian. To be specific, the most significant byte will be exchanged with the least significant byte.

**calldata instruction.** EVM executes the `calldata` instruction to load the input into memory. We maintain calldata in an LLVM array, denoted as $d$. We convert `calldata` to a memory copying operation, coping data from $d$ to $v$.

**storage instruction.** `sload` and `sstore` can use the persistent data in the smart contract storage. We create a hash table $\sigma$ as the storage. Their operand can be obtained from the simulating stack.

**computation instruction.** `op` is a group of instructions used for bitwise and arithmetic operations. For a binary EVM instruction, it has to pop two stack items as operands, and hence we use the two memory data $\mu[p]$ and $\mu[p-1]$ as LHS and RHS values, respectively. Then, the execution result can be pushed to the new stack top $\mu[p-1]$.

**sha3 instruction.** `sha3` is used to compute a hash string in EVM. To emulate it, we implemented a Keccak256 [4] hash function in the GPU environment. The GPU-based Keccak256 function utilizes only thread-local memory, such as local memory and registers.

**call instruction.** `call` is an abstract instruction. Each instance instruction is related to cross-contract calls, i.e., `CALL`, `STATICCALL`, `DELEGATECALL`, `CREATE`, and `CREATE2`. To simulate the cross-contract calls in GPU, we convert `call` to an LLVM function call. The function parameters come from the simulating stack $\mu$. At default, the return value is set to true to simulate a successful call. Since we have no external contracts in the GPU environment, the GPU's support for cross-contract calls is limited. It disables the actual cross-contract interactions within the GPU context. Nonetheless, MAU retains its efficacy in fuzzing cross-contract interactions because: (1) MAU provides full support for cross-contract calls on the CPU side; (2) although the inaccuracies due to incomplete support for cross-contract calls may result in redundant seeds on the GPU, these seeds will be validated in a full-fledged EVM on the CPU side; (3) despite this limitation, the GPU's high throughput empowers MAU to identify more bugs at a faster pace when compared to baselines (see RQ2).

**getenv instruction.** `getenv` is an abstract instruction. Each instance instruction gets the current blockchain environment data. A `getenv` instruction typically pushes a certain value onto the EVM stack, such as blockchain id, block hash, timestamp, block height, block difficulty, block's beneficiary address, gas limit, gas price, remaining gas, account balance, execution origination address, caller address, the deposited value from the caller, smart contract address, and smart contract bytecode. We transform the blockchain environment data into global variables, with `getenv` instructions retrieving the relevant global variables as the resulting output. In particular, we use a constant array to store the EVM bytecode because smart contracts cannot modify bytecode.

**control instruction.** Smart contracts in EVM can exit via `stop` or `suicide` instruction. We represent this with a `return` instruction in LLVM IR. In particular, we move the balance of the executing smart contract to the address of $\mu[p]$, according to the EVM specification. In addition, `revert` can abort the execution and throw an error to EVM. We translate `revert` to an exit instruction which returns a Boolean value indicating the exit code of the smart contract. Note that we translate `revert` to a gentle stop without raising a sticky error, because the sticky error will terminate other threads that are expected to continue, resulting in a decrease in throughput. Regarding `br` and `bne`, we explain them in § 4.1.3.

**4.1.3. Resolving Jumps.** Once the LLVM IR in basic blocks is established, we identify the jump destinations of `br` and `bne`. Since `br` in EVM is an indirect jump that takes the destination from the top item of the stack, we lift it to a table jump (i.e., `switch`) with the complete jump table constructed in § 4.1.1. The destination is emulated by an LLVM memory data, i.e., $\mu[p]$. Although such a jump table results in big "switch" instructions, the evaluation in Appendix A shows that it only increases execution time by approximately 8%, which is a relatively low overhead penalty. To reduce overhead, we can use pointer analysis to precisely identify each jump's individual jump table. This should result in a smaller jump table because each jump has limited branches. In addition, `bne` is a conditional jump that *"falls through"* to the next instruction when the condition is unsatisfied. We can identify its *"falls through"* destination rather than leveraging the jump table because the *"falls through"* edges are explicit in EVM bytecode.

To create a GPU-friendly application, we infer the destinations of indirect jumps and convert them to direct jumps whenever feasible. With the identified jump destinations, GPU can leverage branch prediction to be efficient and ease the branch divergence [13]. Specifically, we perform a backward search [35] to identify the stack top item used by `br` and `bne`. Taking `push` $c$`;br` as a basic block example, we can infer its successor is $c$ because the stack top of `br` must be $c$.

**4.1.4. EVM Exception Handling.** EVM costs gas whenever it interprets an EVM instruction. In addition to executing `suicide`, `stop`, and `revert` instructions to exit explicitly, Ethereum smart contracts implicitly terminate themselves when the remaining gas is insufficient to execute more instructions. The official Ethereum [19] calculates gas usage as the sum of per-instruction gas consumption for all instructions within each basic block. We followed that to estimate gas deduction. To this end, we create a register representing the total gas amount to recover this behavior. In each LLVM basic block, we deduct a certain number of gas units from the gas register to simulate the gas cost in EVM. The gas cost of each basic block is the sum of gas used by the EVM instructions lifted in the basic block. A conditional branch stops the execution when the value of the gas register is lower than zero.

## 4.2. Vectorizing LLVM Assembly

EVM instructions are emulated by LLVM memory operations (§ 4.1). According to the data lifecycle, we rewrite $\mu$, $v$, $d$, and $\sigma$ for code vectorization. The thread assigned with the thread id $i$ can access its thread-safe components, i.e., $\vec{\mu}_i$, $\vec{v}_i$, $\vec{d}_i$, and $\vec{\sigma}_i$. Memory operations are friendly to code vectorization because we can convert a memory operation to a vector operation by adding a linear address. To enable the smart contract to obtain the thread id in GPU runtime, we import a built-in API from the PTX LLVM backend to the PTX smart contract.

**4.2.1. Local Data.** We use local memory to emulate the stack, memory, and calldata because their data will be cleared immediately after the smart contract execution finishes. The local memory of different threads has different address spaces. Thus, operations to local memory are naturally thread-local and can support SIMD execution with minor efforts. Specifically, two arrays allocated in local memory are used in the emulated stack and memory. The local memory is allocated via `malloc` at the entry of the smart contract. To reduce memory consumption, we free the allocated memory once a thread finishes. In the SIMD runtime, threads can execute the same `malloc` instruction together but obtain a different value as the pointer, which points to the thread-local memory serving as the EVM stack. Following the EVM specification, each item in the emulated stack has 32 bytes, and both the emulated memory and calldata are byte-addressing arrays.

**Algorithm 1:** Vectorizing storage instructions.

```
Global: snapshot $n_i$; master volume $m$
Input : storage index $x$; storage value $y$
1  Function SSTORE($x, y$):
2      if $x \in \{slot \mid slot \in n_i\}$ then
3          $n_{ix} \leftarrow y$
4      else
5          $n_i \cup \{x : y\}$

6  Function SLOAD($x$):
7      $y \leftarrow 0$
8      if $x \in \{slot \mid slot \in n_i\}$ then
9          $y \leftarrow n_{ix}$
10     else
11         if $k \in \{slot \mid slot \in m\}$ then
12             $y \leftarrow m_k$
13     return $y$
```

**4.2.2. Persistent Data.** Smart contracts can execute `SSTORE` and `SLOAD` to access persistent data in the key-value storage. In our fuzzing, seeds run together and share the same initial storage content. To reduce memory consumption, we build a redirect-on-write snapshot [28] as the EVM storage, which includes one master volume and multiple incremental snapshots, denoted as $m$ and $\vec{n}$, respectively. In other words, $\vec{\sigma}_i$ is an abstract object from ($m$, $\vec{n}_i$). Each element represents a storage slot in the EVM. A slot is a structured data with two 32-byte data. The first 32 bytes represent the storage key, and the second 32 bytes represent the corresponding storage value. The lightweight storage structure can be used as the hash mapping defined in the EVM specification. $m$ maintains the initial content. When running SIMD threads, the storage used by thread $i$ is an incremental snapshot from $m$. Since new data writes are redirected to $n_i$, all GPU threads can share $m$ as the initial storage content. To be specific, we rewrite `SSTORE` and `SLOAD` as shown in Algorithm 1.

- `SSTORE(x, y)` stores `y` in the storage using `x` as the hash key. To handle this, we redirect the written data (i.e., $y$) to snapshot $n_i$. Here, $n_{ix}$ represents the value in the thread storage $n_i$ where $x$ is the hash key.
- `y = SLOAD(x)` loads `y` from the storage using `x` as the hash key. To handle this, we first search $x$ from $n_i$. If it does not exist, we then search $m$. Otherwise, we will eventually return zero to fulfill the EVM specification.

## 4.3. Instrumenting Smart Contract

To evaluate seeds running in SIMD threads, we instrument an LLVM vector. Each element in the vector is a bitmap. Each thread uses its thread ID to fetch a bitmap from the vector. Individual threads can track the execution state independently. In addition to increasing code coverage, seeds that satisfy transaction dependency should be prioritized for testing. To this end, we extend the bitmap format to keep track of the branches that have already been seen and the dataflow dependency (read-after-write) between state variables. We utilize the data dependency between state variables to resolve transaction dependencies when feasible. In the following section, we first explain why the edges-only bitmap (i.e., AFL [68]) is ineffective. Then, we propose the stateful bitmap to ease the transaction dependency. Finally, we use a parallel algorithm to count the interesting seeds.

**4.3.1. Edges-only Bitmap.** AFL defines a tuple $(src, dst)$ to record one branch transition, i.e., the control flow edge from one basic block ($src$) to another one ($dst$). When using AFL bitmap, $b_i$ indicates the bitmap for the thread $i$. The bitmap byte in thread $i$ like $b_{ij}$ is the hit count for the taken branch $j$. The branch index $j$ is the hash from $(src, dst)$:

$$j = src \ll 1 \oplus dst$$

After the thread $i$ finishes, the number of non-zero bytes in $b_i$ is the number of distinct branches the seed hits. If more branches are explored, fuzzers mutate this interesting seed again regarding the increased code coverage.

However, due to transaction dependency, the code coverage metric is less effective in evaluating the input seeds of smart contracts. We use Figure 1 as the example to explain the reason. Suppose we have the following four seeds:

$S_0$ : `vul(0)` $S_1$ : `dep(0)`

$S_2$ : `dep(2)` $\rightarrow$ `vul(0)` $S_3$ : `vul(0)` $\rightarrow$ `dep(2)`

After $S_0$ and $S_1$ are evaluated, the coverage-based fuzzer will take $S_2$ as an uninteresting seed because it covers the same branches that $S_0$ and $S_1$ have already covered. But $S_2$ is worth testing more because it has an expected transaction sequence, i.e., $dep \rightarrow vul$. `dep` can change the state variable $y$; `vul` can make the branch at line 4 satisfied. Therefore, seeds with an expected transaction sequence should also be interesting, even though they may not increase code coverage.

**4.3.2. Stateful Bitmap.** Our feedback metrics include the code coverage and new data dependency of state variables. Evolved from the AFL bitmap, a stateful bitmap consists of branch hits and the changes of state variables. To be specific, the bitmaps index is a hash:

$$j = hash(src, dst, stat) = (src \ll 1 \oplus dst) \oplus stat$$

The first part (i.e., $src \ll 1 \oplus dst$) keeps track of the hit counts of already-seen branches. The second part (i.e., $stat$) is a global register that maintains the current state of the smart contract. Whenever a state variable $val$ is loaded by a $val \leftarrow$ `SLOAD(key)`, $stat$ hosts the state with a hash value, i.e., $stat \leftarrow key \oplus val$. $stat$ is initiated to zero, thus if no `SLOAD` is executed, the bitmap hash degenerates to the vanilla one, i.e., $src \ll 1 \oplus dst$. Therefore, we believe a potential new data dependency between state variables exists, whenever one transaction loads a state variable changed by previous transactions. Even if no new branches are explored, the seed's value will increase due to the found transaction dependency, and the seed will become interesting.

The stateful bitmaps can find the interesting seed used in Figure 1. Although $S_2$ hits no new branches, the seed's value still increases due to $stat$ changing from 0 to 2. $S_3$ remains uninteresting because $stat$ used in `vul(0)` remains zero resulting in the same seed's value as $S_0$ and $S_1$ have. Therefore, the fuzzer can choose $S_2$ again and mutate more, i.e., changing the function arguments of its two transactions. New interesting seeds will be generated whenever the first transaction argument changes to a new value. If the value of `y` is set to the expected value, i.e., 4, then the second transaction, i.e., `vul(0)`, can hit more branches, making the entire seed interesting. Therefore, the interesting seed taking `dep(4)` as its first transaction will be mutated again until the expected argument of `vul()` is found as well.

The stateful bitmap allows MAU to track transaction dependency effectively and efficiently. Although the fuzzer has to test more seeds for complex transaction dependency, the bitmap ensures the testing of transactions with read-after-write dependencies among state variables. It fulfills our design target, as fuzzing is insensitive to the number of seeds because of the high throughput. As long as the expected seeds are added to the seed queue successfully, fuzzers still have an opportunity to trigger the vulnerability later.

**4.3.3. Parallel Counting.** We use $N$ as the total thread number. We have $N$ bitmaps in total, denoted as $b_0$ ... $b_{N-1}$. Each bitmap should be large enough to reduce the hash collision rate. Thus, we count all bitmaps in parallel to boost fuzzing.

First, we encode all bitmaps into an incremental bitmap named `virgin_bits` [68], denoted as $B$. The byte (i.e., $B_j$) in `virgin_bits` represents whether the corresponding branch has been explored by any thread yet or not. Initially, the entire `virgin_bits` is fully masked, i.e., each byte value is 0xFF. We set $B_j$ to zero if we found the branch $j$ was hit in thread $i$, i.e., $b_{ij} \neq 0$. If $B_j$ remains 0xff in the end, Evaluator deems the branch unexplored by any seed.

Second, we split $B$ and $b_i$ into $\frac{\epsilon}{8}$ parts in eight bytes and distribute each piece to one thread, where $\epsilon$ is the bitmap size. We set the window size to eight bytes because the GPU is 64bit. After the smart contract execution, we can use additional $\frac{\epsilon}{8}$ threads to encode $b_0$ ... $b_{N-1}$ to $B$. Each thread $i$ updates the portion of $B$ starting from the index of $8i$ with a size eight bytes. The formal description is shown below.

$$B_j \leftarrow B_j \& \neg b_{kj}, 0 \leq k < N, \quad i \leq j < i+8, 0 \leq i < \frac{\epsilon}{8}$$

A seed is interesting if there are non-zero bytes counting in the `virgin_bits`: $\sum_{i=0}^{\epsilon} B_i \neq 0$

## 4.4. Detecting Bugs

MAU replays seeds in the CPU environment to detect bugs. To detect bugs in smart contracts, which do not typically result in crashes, we require oracles. Below are the design specifics for the four most prevalent bug types.

**IB.** Integer Bug (**IB**) is caused by integer overflows or underflows, making the arithmetic result become an unexpected value. To detect it, we monitor the results of arithmetic instructions (i.e., `op`). If the result value is outside the expected range, we report a bug.

**BD.** Block state dependency (**BD**) occurs when a smart contract uses the state of a block (i.e., `getenv`) to decide the Ether transfer of a contract. Attackers can bypass the block state checks to withdraw the victim's cryptocurrency. To detect it, we first track the data flow of block state variables and then monitor if the operands of a `call` depend on the block states.

**RE.** Reentrancy (**RE**) occurs when a re-entered vulnerability leads to a race condition on state variables [63]. To detect it, we monitor the invoked cross-contract calls (i.e., `call`) and identify state variables that affect these calls. We report a bug if such state variables are updated after the calls take place.

**ME.** Mishandled Exception (**ME**) occurs when a contract does not check the return value of a cross-contract call. We detect it by observing the return value of the `call`; if it remains unchecked by subsequent instructions, we flag a bug.

## 4.5. Hybrid Mutation

Mutation introduces small changes to the incoming seed that may still keep the input valid, yet exercise new smart contract behavior. To reach high seed diversity, we design a hybrid mutation mechanism on both CPU and GPU. On the one hand, CPU-side mutation generates one seed to GPU, while GPU-side mutation generates a batch of seeds for all threads. GPU mutation is to avoid frequent data transfers between CPU and GPU, which would otherwise reduce throughput. On the other hand, we enable various mutation strategies to mutate the seeds, such as bitflip and havoc. CPU mutates both transaction sequence and arguments, while GPU mutates arguments only. Note that GPU does not change the transaction sequence for easing branch divergence. Threads would execute different branches from the start and cause overhead due to branch divergence, if they run different smart contract functions together.

TABLE 3: The size of the two benchmarks.

| | #Cnt | Src/LoC | | #Func | | Bytecode/KB | |
|---|---|---|---|---|---|---|---|
| | | mean | std. | mean | std. | mean | std. |
| Small | 130 | 136 | 121.20 | 10 | 8.30 | 4.35 | 3897.15 |
| Large | 400 | 13099 | 2961.28 | 406 | 117.70 | 13.95 | 8328.88 |

## 4.6. Implementation

In order to evaluate the design of MAU, we implemented a prototype using approximately 8,000 lines of C++ code and 100 lines of Rust code. The prototype implementation has been released.

**Transformer.** To align with EVM specification [64], we configure $\vec{\mu}_i$ as a *1024x32* bytes vector, set the size of $\vec{v}_i$ to 1448B, and establish the size of $\vec{d}_i$ to 2048B. Each abstract storage is organized as an array with 32 slots. To support SHA3 in GPU, we compile a Keccak256 algorithm from C language to LLVM assembly and then link it with the smart contract.

**Vectorizer.** To reduce the hash collision in bitmap, each thread uses a 4KB bitmap. Users can configure the above parameters to meet their requirements. We translate the LLVM assembly to a 64-bit PTX binary through the LLVM PTX backend with all optimizations rules enabled (i.e., *clang -m64 –target=nvptx64-nvidia-cuda -O3*).

**Evaluator.** We implemented a hybrid fuzzer based on Ityfuzz [54]. The GPU extension is powered by the low-level CUDA driver [48] (v11.3). For fuzzing, 1,024 threads are executed on a single GPU card.

## 5. Evaluation

Our evaluations are driven by the following research questions (RQs):

- **RQ1**: Does Transformer obtain the semantic correctness in bytecode translation?
- **RQ2**: How effective is MAU compared to state-of-the-art tools?
- **RQ3**: Do MAU's components contribute in improving performance?
- **RQ4**: Can MAU uncover new bugs in real-world smart contracts?

## 5.1. Experiment Setup

**Benchmarks.** Our experiments run on two benchmarks, including a large benchmark and a small one (see Table 3). Here are the details of the benchmark construction. Before January 2023, Etherscan had released 183,111 smart contracts with verified source code [50]. To evaluate fuzzing performance on complex smart contracts, we construct a large benchmark consisting of smart contracts with more than 10,000 lines of source code (LoC). After identifying 618 complex smart contracts, we removed duplicates, resulting in a final set of 400 smart contracts for the large benchmark. During the experiments, we found that the existing tools have scalability issues in the large benchmark. For a fair comparison, we further construct a small benchmark sourced from existing work, including 58 smart contracts with assigned CVEs [56] and an additional 72 curated from Smartbugs [15]. In particular, each smart contract in the small benchmark is a ground truth for the following bug types: integer bug (**IB**), block state dependency (**BD**), mishandled exception (**ME**), or reentrancy (**RE**). We chose these bug types because the state-of-the-art tools support them.

The large dataset has 400 smart contracts with an average of 13,099 LoC, defining approximately 406 functions each. In contrast, the small dataset consists of smart contracts with an average of 136 lines of source code, defining around ten functions per contract. The source code size of the large benchmark is nearly 100 times greater than that of the small benchmark. Note that we collect source code for manual reasoning in case studies. MAU can test closed-source smart contracts with only their bytecode and ABI [8]. Throughout the experiments, no tool relied on the source code of smart contracts.

**Baselines Setup.** We first looked for open-source tools that are published in top conferences and obtained Ityfuzz [54], Smartian [11], ContractFuzzer [32], ILF [27], sFuzz [47], Mythril [46], and EF\CF [52]. We excluded Harvey [66] because its authors did not release the tool to the community. Echidna [25] and ConFuzzius [62] were also omitted from the experiments since they cannot analyze closed-source smart contracts. We chose Ityfuzz, Smartian, ILF, and sFuzz over ContractFuzzer based on the findings in [11], which demonstrated their superior performance. Although EF\CF aimed to enhance throughput, we opted for Ityfuzz for its superior speed, and thus excluded EF\CF from consideration. All baselines use the default setting provided by their artifacts. To gain a better understanding of the contribution of the GPU parallel mutation (GPM) and the stateful bitmap feedback (SBF), we performed an ablation study by comparing MAU with MAU-G (MAU without GPM) and MAU-S (MAU without SBF). More specifically, MAU-G mutates seeds together on 32 CPU cores; MAU-S adds the seeds to the corpus only when the execution hits new instructions.

**Environment Setup.** All experiments were conducted on a server running Ubuntu 20.04 LTS equipped with dual Intel Xeon Gold 6226R processors (32 cores), 512 GB RAM, 4T HDD, and five NVIDIA RTX3090 graphic cards (10,496 CUDA cores and 24 GB VRAM for each GPU device). We use instruction coverage (i.e., the number of distinct instructions explored during the testing) as the metrics of the code coverage. Instructions from the constructor and view functions were not included in the coverage analysis because of the following reasons. The constructor of the smart contract is out of the attacker's control, and the view functions are designed to run locally on the callers' side without causing any impact on the online blockchain data. For the code coverage and bug detection experiments, we ran all tools for one hour. This timeout is selected because we observed that all tools can reach the maximum code coverage within the first 20 minutes (see RQ2). Each experiment set was run within a clean Docker container to minimize environmental interference.

## 5.2. Correctness Verification

Transformer is essential to code translation. To determine that the PTX code remains consistent with the original EVM smart contract, we employ a differential testing involving, collecting ground truth test cases that can trigger **IB**, running test cases on both CPU and GPU, and comparing the bug locations between EVM bytecode and PTX code. To enable the PTX code in GPU to output the bug location to CPU's stdout, we import an **IB** sanitizer to Transformer. Thanks to the **IB** sanitizer released in the LLVM ecosystem, we can import it without introducing bias due to handcrafted sanitizers. The generated LLVM assembly of Transformer is translated to PTX here, skipping both code vectorization and instrumentation. We only choose **IB**, because RQ1 aims to assess translation accuracy, not oracles. We use the oracle only to record smart contract outputs for validation against the ground truth, focusing solely on IB since Verismart's ground truth pertains to **IB**. In the end, we use 58 smart contracts with assigned CVE identifiers and collect 317 test cases from [56] for verification.

**Results.** Among the 58 smart contracts, Transformer can successfully translate 100% of them. The PTX code passes all test cases and outputs the expected program points where an **IB** has been triggered. This experiment has also been used to improve our implementation. When the input buffer (e.g., calldata) of the GPU was initially set to 128B, 65 test cases from 17 smart contracts failed because 128B is too small to store the input coming from the CPU. We solve this issue by configuring the input buffer length to 2048 bytes. In the following experiment, we keep this setting. Note that we did not guarantee 100% correctness for the translation of all smart contracts. During fuzzing, Evaluator will replay all interesting seeds found in GPU on CPU for verification.

**Answer to RQ1:** Transformer can achieve high semantic correctness in translating EVM bytecode.

## 5.3. Comparison against Existing Tools

To assess the effectiveness of MAU, we apply it along with all baselines to analyze both the large and small benchmarks and then measure code coverage, the number of detected bugs, throughput, economy, and power consumption.

Firstly, to measure code coverage in terms of instruction coverage, we replay all interesting seeds on an instrumented CPU-end EVM. Secondly, fuzzing typically has no false positives, but the insufficient rules in oracles may lead to incorrect reports. In the large benchmark, we manually verify the bug alarms and classify them as true positives (TP) and false positives (FP). To minimize bias during manual verification, we first locate potential bug locations in the source code based on the fuzzer's concrete exploits; then

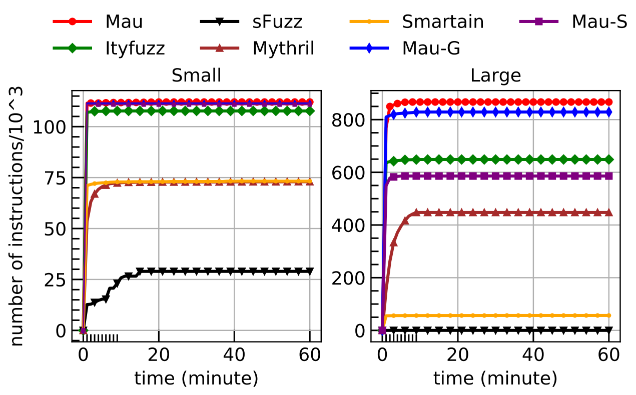


Figure 6: Comparison against state-of-the-art tools on both large and small benchmarks.

TABLE 4: Bug detection on the small benchmark.

| Bug | MAU | | Smartian | | sFuzz | | Mythril | |
|---|---|---|---|---|---|---|---|---|
| | TP | FP | TP | FP | TP | FP | TP | FP |
| IB | 56 | 0 | 53 | 0 | 8 | 0 | 7 | 0 |
| BD | 10 | 0 | 11 | 0 | 10 | 0 | 8 | 0 |
| ME | 47 | 0 | 48 | 0 | 29 | 6 | 46 | 0 |
| RE | 15 | 1 | 19 | 0 | 5 | 20 | 19 | 38 |

TABLE 5: Bug detection on the large benchmark.

| Bug | MAU | | Mythril | | MAU-G | | MAU-S | |
|---|---|---|---|---|---|---|---|---|
| | TP | FP | TP | FP | TP | FP | TP | FP |
| IB | 112 | 0 | 46 | 0 | 104 | 0 | 103 | 0 |
| BD | 54 | 0 | 28 | 0 | 54 | 0 | 28 | 0 |
| ME | 108 | 0 | 59 | 0 | 108 | 0 | 99 | 0 |
| RE | 11 | 3 | 1 | 12 | 10 | 3 | 7 | 3 |

identify TP at the source code level if the exploit indeed can cause the bugs defined in § 4.4. The source code traceback is feasible because the Solidity compiler can generate a mapping from the bytecode to the range in the source code that generated the instruction. As for the small benchmark, we report TP if the tool can pinpoint the exact program address for the assigned bug. Each bug in the small benchmark has a ground truth label indicating the bug location. Thirdly, we evaluate MAU's parallel ability by measuring its throughput, specifically using seed throughput, which is the number of seeds evaluated within one second. Finally, we further estimate the cost effectiveness and energy efficiency of all tools, showing MAU can achieve a greater efficiency—evaluating the same number of seeds using fewer resources. We accomplish this using throughput per dollar, cost per bug, and power consumption.

**Code Coverage.** MAU outperformed all baselines in terms of code coverage, exploring 23.36%-371.43% more distinct instructions than different baselines. Figure 6 illustrates the overall results on both small and large benchmarks. The x-axis represents the fuzzing time in minutes, and the y-axis represents the cumulative instruction coverage.

In the large benchmark, MAU surpassed Ityfuzz and Mythril by covering 33.86% and 75.56% more distinct instructions, respectively. In particular, Ityfuzz, the best baseline in the large benchmark, explored 880.52k instructions before the timeout. In terms of the time to code coverage, MAU is 20.9X faster than Ityfuzz, reaching Ityfuzz's maximum code coverage in just 40 seconds. We did run both sFuzz and Smartian on the large benchmark, however they yielded extremely low code coverage (e.g., 0 and 125 instructions per contract, respectively). sFuzz failed to generate fuzz drivers for the complex smart contracts. Smartian analyzes transaction dependencies from the bytecode's CFG and DFG before launching fuzzing on transaction sequences. However, its static analysis fails due to path explosion in extensive benchmarks. Although we could run Smartian in a dummy mode (static analysis disabled) or allocate more RAM/Time, it would be unfair to other tools.

The further comparison of MAU with baselines in the small benchmark revealed that MAU covered 4.67%, 300.00%, 55.56%, and 53.42% more distinct instructions than Ityfuzz, sFuzz, Mythril, and Smartian, respectively. While the performance improvement of MAU diminished in the small benchmark compared to the large benchmark, we attribute this to the GPU boost being particularly effective in fuzzing complex smart contracts. Small programs with limited input space still allow conventional tools running on CPUs, like Ityfuzz, to achieve high code coverage. Nevertheless, MAU has demonstrated its effectiveness in testing complex smart contracts.

We evaluated the additional time required for these tools to achieve new coverage, running tests on both benchmarks for 12 hours. On the small benchmark, both MAU and Ityfuzz need more time to achieve new coverage. On the large benchmark, MAU obtains higher code coverage after about 6h57m whereas Ityfuzz needs more time.

**Bug Detection.** MAU outperforms baseline methods in detecting bugs with greater accuracy. Table 4 presents the bugs detected by tools in the small benchmark. Ityfuzz is excluded from the comparison because it requires developers to implement bug oracles at the source code of the target smart contract. The experimental results show MAU found 1.01X, 2.5X, and 1.65X more bugs than Smartian, sFuzz, and Mythril, respectively. In terms of discovery speed, MAU is faster than the baselines. During testing, MAU detected 128 TPs after just one minute of fuzzing. In contrast, Smartian and sFuzz find fewer bugs (112 and 46, respectively) within the same timeframe. Table 5 illustrates the results on the large benchmark. We exclude Smartian and sFuzz here as they both fail to analyze the complex examples. MAU found 2.2X more bugs than Mythril. All bugs detected by the Mythril can be found by MAU.

**Bug Analysis.** All these tools, including MAU, missed the detection of CVE-2018-13325 and CVE-2018-13695. CVE-2018-13695 has an integer bug in the function `mint(address receiver, uint amount)`. To trigger the bug, the input must ensure the argument `receiver` is equal to the specific ad-

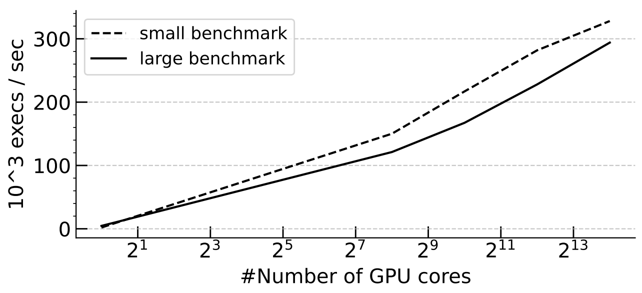


Figure 7: MAU's throughput over the number of GPU cores. The dashed line indicates the throughput when running the small benchmark. The bold line indicates the throughput when running the large benchmark.

TABLE 6: Comparison against state-of-the-art fuzzers on the aspects of financial expense and power usage. For each benchmark, there are three columns, such as throughput, throughput per dollar, and power consumption.

| | small benchmark | | | large benchmark | | |
|---|---|---|---|---|---|---|
| | $10^3$exec/s | $10^6$exec/$ | J | $10^3$exec/s | $10^6$exec/$ | J |
| **Smartian** | 0.66 | 16.97 | 461.71 | N/A | N/A | N/A |
| **Ityfuzz** | 21.33 | 548.49 | 14.30 | 18.68 | 480.34 | 16.31 |
| **MAU** | 328.06 | 4724.06 | 7.90 | 162.37 | 2338.13 | 16.02 |

dress. One potential solution is a dictionary-based mutation, which collects magic numbers/constants from the smart contract bytecode and then uses this corpus to construct new seeds. CVE 2018-13325 has a bug in `transfer(address this, uint256 _value)`. It requires the exploit to have a hard-to-resolve transaction sequence. Its `transfer()` function can call two sub-functions such as `_sell` and `_transfer` depending on its first function parameter, i.e., `this`. To trigger the bug, the attacker has to first execute `transfer` to use `_transfer` to set state variables, and then execute `transfer` again but to use `_sell` with expected function arguments to trigger an integer overflow. In other words, we must invoke `transfer` twice and control the parameter to execute the expected sub-functions in sequence. MAU incurs this false negative due to the overlap of the stateful bitmap. It may fail to identify some valuable transaction sequences.

MAU also incurs false positives due to the insufficient rules in the **RE** oracle. In the contract with address 0x05be...9583f, a function is used to initiate itself, configuring the Ether router. To hit the bug, we have to set the router to a specific contract address to control Ether transfer and launch the reentrancy attack. However, the router can only be configured at the initiation function, which cannot be executed by the attacker.

**Throughput.** As shown in Table 6, MAU achieved an overall throughput of 162.37K execs/sec and 328.06K execs/sec in the large dataset and small dataset, respectively. This is 8.69X and 15.38X faster than Ityfuzz. When being compared with Smartian, which achieves 0.66K execs/sec on the small benchmark, MAU is 497.06X faster. Even when run in a single thread, MAU outperforms Smartian with a throughput of 2.04K execs/sec, which is approximately 3.09X faster.

TABLE 7: The pure execution time of tools in evaluating 1024*32 random seeds. We measure the average time used for all smart contracts in each benchmark.

| | small benchmark | large benchmark |
|---|---|---|
| **Smartian** | 37143.7ms | N/A |
| **Ityfuzz** | 1212.4ms | 1605.6ms |
| **MAU** | 121.6ms | 156.8ms |

Since our goal is to accelerate Step ② of fuzzing (i.e., seed evaluation), we measure the pure program execution time for each tool running 1024*32 random seeds (See Table 7). In the small benchmark, MAU spends 121.6ms in running 1024*32 random seeds, significantly outpacing both Smartian and Ityfuzz. In the large benchmark, MAU remains more efficient than the best baseline, i.e., Ityfuzz. The results show that we successfully improved throughput by boosting Step ②.

M. Böhme and B. Falk found that finding new bugs with fuzzing needs an exponential increase in the number of computation resources, i.e., CPU cores [6]. To investigate the relationship between cores number and throughput, we configure MAU to use different numbers of cores from 1 to $2^{14}$. In Figure 7, we show the average throughput of MAU over the number of GPU cores. The dashed and solid lines illustrate the increase in throughput when testing the small and large benchmarks, respectively. By exponentially increasing the number of GPU cores, the throughput has a semi-linear increase over the number of GPU cores. It shows a potential improvement in throughput by increasing the core number. For example, we can launch MAU on multiply GPU cards together to boost the throughput further.

**Throughput per Dollar.** In Google-Cloud, 1,024 GPU-cores on a Tesla-T4 ($0.11/h; 70W) are more cost-effective than one CPU core ($0.14/h; 9.3W). As RTX3090 is unavailable in Google Cloud, we use Tesla-T4 to estimate the throughput per dollar (exec/$). In both benchmarks, MAU is more cost-effective than Smartian and Ityfuzz. In the small benchmark, compared to Ityfuzz and Smartian, MAU packs 8.6X and 278.45X more throughput-per-dollar, respectively. In the large benchmark, MAU is 4.9X more efficient than Ityfuzz. Smartian is excluded from the large benchmark evaluation due to its scalability issues.

**Cost per Found Bug.** We also compare the cost per found bug. On the small benchmark, MAU needs around $0.25 to detect one bug, which has no distinguish improvement against Smartian ($0.14), sFuzz ($0.35), and Mythril ($0.22). On the large benchmark, MAU is 52% cheaper than Mythril.

**Power Consumption.** We require each tool to run 1024*32 random seeds and we estimate their end-to-end power consumption. As shown in Table 6, although MAU requires an additional GPU device, it is more energy-efficient than baselines.

**Answer to RQ2:** MAU outperformed state-of-the-art tools in terms of performance, power consumption, and economy.

## 5.4. Ablation Study

To gain a better understanding of the contribution of GPM and SBF, we compare MAU with MAU-G and MAU-S on the large benchmark.

**Results.** As shown in Figure 6, MAU finds 3.83% and 1.74% more instructions than MAU-G and MAU-S, respectively. Compared to MAU, MAU-G requires more time for seed mutation because of the slow seed transfer between CPU and GPU. As for MAU-S, it tends to test duplicate seeds for more mutations, because without SBF the fuzzer is less effective in identifying transaction dependencies and thus misses valuable seeds. In terms of bug detection (see Table 5), MAU found more bugs than MAU-G and MAU-S.

**Answer to RQ3:** Both GPM and SBF contribute to MAU in improving performance.

## 5.5. Identified New Bugs

To demonstrate the capability of MAU in uncovering new vulnerabilities, we use MAU to test all 183,111 real-world smart contracts. The timeout is set to ten minutes to ensure that we can complete this experiment within a reasonable time. To identify new bugs, we exclusively enable the fund-leaking oracle while disabling all other bug detection oracles, because the fund-leaking oracle can automatically detect bugs that can result in financial loss. Specifically, MAU executes the seeds identified as interesting by the GPU on the CPU to check against the fund-leaking oracle. This oracle flags bugs when the attacker's balance increases post-seed execution. Any seed triggering the oracle represents a transaction sequence that can potentially siphon cryptocurrency from a victim in EVM.

**Results.** MAU successfully detected vulnerabilities in 548 projects, identifying potential exploits in cryptocurrency assets worth over $172K[3]. MAU uniquely identified 119 bugs that were not detected by any of the baseline tools. Ityfuzz identified 423 bugs, requiring a 12.7% longer fuzzing duration to do so. Ityfuzz missed 122 bugs even after a 24-hour timeout. Smartian failed to find 413 bugs identified by MAU. Mythril and sFuzz did not detect any bugs that can cause Ether leaks.

Out of 548 bugs, 93 victims contain a public method that can initiate themselves. Attackers can front-run the victim's creator to obtain sensitive permission for a successful exploit. We performed a detailed manual investigation on a random subset of 50 contracts selected from the remaining 455 reported alarms. There are 21 Ponzi games and 20 lottery projects in which attackers can gain more cryptocurrency than what they invested. In addition, MAU can trigger arbitrary external calls and arbitrary ERC20 token burning in nine smart contracts. To estimate the potential damage, we compile comprehensive statistics on affected entities, including project names, creator addresses, and each victim's total assets. Estimating the risk to the assets requires manual analysis, which is part of our future work. The detailed statistics can be found in artifact.

3. cryptocurrency price is from Etherscan ($2374/Ether)

Due to the anonymity of Ethereum, tracing smart contract creators is often infeasible, which complicates bug reporting to the respective developers. Therefore, we have submitted these issues to the CVE database for further confirmation. Two CVE identifiers have been assigned, such as CVE-2024-28260 and CVE-2024-28261. Next, we present case studies to illustrate the bugs.

**Arbitrary Transfer.** In the smart contract with address 0xCE5093Dd7cf90699Bba881af8f2c8aD0A7066dC5, the attacker can execute the public function `multiTransfer()` to transfer an arbitrary amount of Ether from the victim to anyone. To confirm the bug, we launch the attack in a forked chain and successfully obtain the victim's entire balance.

**Ownership Backdoor.** In the smart contract with address 0x24a7de87b3bd7298bbf8966fdf170c558d69ecc4, MAU discovered a backdoor that allows an attacker to change the victim's ownership to an arbitrary address. Figure 8 shows the simplified source code of the victim, METADOLLAR. `Owned` library is utilized for ownership management, and METADOLLAR inherits from `Owned`. Consequently, METADOLLAR's constructor is automatically executed with `Owned` to initialize the owner as its creator. However, `owned()` is not the constructor of `Owned` because the Solidity v0.4.18 compiler requires the constructor function to have exactly the same name as the contract name. In L5, the function `owned()` has a case mismatch with `Owned`, thus `owned()` becomes a public function that anyone can invoke. Once the ownership is obtained, the attacker can withdraw all the victim's funds by executing `collect()`.

**ERC721 Reentrancy.** The smart contract in 0x4aeaf7ddb924bfb19d7ff205de7893c5dd429288 is an NFT market for users to mint and transfer tokens in the ERC721 standard. The function createVestForDeposit() mints an NFT (_safeMint) and sends a reward to the caller based on the current NFT supply; However, the supply amount is updated after the _safeMint(), which incurs an external call under the sender's control. The attacker can reenter to mint another NFT, but the reward between the two remains the same, thus the attacker can receive more reward than expected to make a profit.

**Answer to RQ4:** MAU found 548 new bugs alive in real-world smart contracts, holding assets valued at over $172K.

## 5.6. Threats to Validity

Our work faces several limitations. First, potential inaccuracies in code translation may cause ineffectiveness: a) storage size. MAU utilizes a hash table to model EVM storage. If the smart contract's storage exceeds the default size (i.e., 32 slots), storage data on the GPU may be overwritten unexpectedly. To solve it, we can choose a larger parameter. b) cross-contract calls. MAU uses function calls to model cross-contract interactions. Like CPU-based fuzzers, MAU faces limitations with unknown callee addresses and resorts to dummy calls in such cases. A potential solution is to convert cross-contract calls into cross-function calls by (1) identifying the cross-contract call and its callee address; (2) obtaining the bytecode of the callee's address and translating it into an individual PTX library; (3) converting the smart contract calls to library calls. c) contract creation. EVM smart contracts can generate new contracts with designated bytecode. As GPUs cannot execute bytecode, MAU converts contract creation into a dummy instruction by only making the stack balance, thereby disabling the actual creation of new smart contracts within the GPU context. To fix it, we can statically identify and translate the smart contracts to PTX format, then call their constructors during GPU execution. Nonetheless, MAU retains its efficacy in fuzzing by providing full EVM support on the CPU side and leveraging the GPU's high throughput to identify bugs faster compared to baselines.

Second, manual verification in the evaluation may introduce bias. As we aforementioned before, we confirm true alarms by matching the tool's report with the source code. In addition, we curated a ground truth benchmark from existing work, in which all bugs are marked with program points. In the future, we will extend the benchmark to evaluate MAU using a larger benchmark with a wider variety of bugs.

Third, one oracle was used in RQ4. We only enable fund-leaking oracle because it can automatically detect bugs that can result in financial loss. Expanding oracle support is achievable through additional engineering. Since MAU is based on LibAFL [17], we offer LibAFL-compatible interfaces for developers to easily add new oracles.

While increasing CPU resources can enhance fuzzing efficiency [67] as well, hardware constraints cap the potential gains. The basic unit of parallelization is the computing core. A single GPU, typically with thousands of cores, provides more parallelism than several CPUs, which usually have only 8 cores each. MAU is more cost-effective and energy-efficient than baseline methods in terms of both financial expense and power usage (See RQ2). Effective parallel fuzzing is difficult due to the need for periodic synchronization between fuzzing instances. CPU fuzzers often pause to synchronize and negotiate a global seed pool. MAU overcomes this by simultaneously collecting bitmaps from all instances and identifying interesting seeds in parallel. Moreover, MAU can incorporate fuzzing strategies available in existing smart contract fuzzers, achieving a better result. MAU is built on top of LibAFL [17], and we enabled various mutation strategies (e.g., bitflip and havoc), seed scheduling, and guiding metrics (i.e., code coverage and dataflow). To integrate MAU with other fuzzing techniques for a better result, users only need to implement a fuzzing stage in Rust using the LibAFL APIs and add the created stage into the main fuzzing loop.

Although GPU can boost parallel execution, there exist some downsides of GPU fuzzing. 1) The GPU's inability to natively execute system calls complicates the conversion of general programs to GPU applications. MAU circumvents this by emulating a blockchain environment on the GPU for EVM instruction support, though this can result in some redundant seeds due to execution inaccuracies on the GPU. 2) The data transfer overhead between CPU and GPU can decrease throughput. MAU minimizes this by embedding mutation and feedback mechanisms directly into the GPU.

In the future, we will explore the feasibility of fuzzing general applications on GPUs. There is still a gap in running general programs on GPUs since GPU applications cannot invoke system calls provided in the CPU host. Nevertheless, MAU took the first step toward GPU-accelerated fuzzing.

## 6. Related Work

### 6.1. SIMD on GPU

SIMD (Single Instruction, Multiple Data) vector organizations in that a single instruction controls a group of processing units named warp. In CUDA, a warp contains 32 threads, which always run together. In every GPU cycle, the same instruction is executed in the 32 threads in the same warp. The GPU maintains an execution state per thread, such as a program counter, operand stack, and local memory, and can yield execution at a per-thread granularity, either to make better use of execution resources or to allow one thread to wait for data to be produced by another.

### 6.2. Smart Contract Fuzzing

Fuzzers execute the target program with the inputs created in a mutational manner, such as blackbox, whitebox, and greybox. As the first-generation fuzzer in detecting vulnerabilities in Ethereum smart contracts, ContractFuzzer [32] used a blackbox mutation to create random test inputs. Although it found several bugs, it failed to explore complex branches and cover more code because blackbox fuzzers easily evaluate a lot of redundant seeds. To search the solution of the complex branch, whitebox fuzzers [27] use symbolic/concolic executor to generate new input seeds systematically. They build a sequence of symbolic bytes as the seed and execute the smart contract to construct the symbolic constraints from the path branches. An SMT solver solves the path constraints and then maps the SMT solution to the seed bytes. However, SMT solving is time-consuming and causes low throughput. Greybox (a.k.a. feedback-guided) fuzzers are another option. [47], [66], [25], [54], [52], [11] defined a fitness function to evaluate seeds and mutate the ones with higher value.

### 6.3. Binary Translation

Binary translation is one common technique to migrate applications from one ISA (Instruction Set Architecture) to a new one without affecting its functionality. Dynamic translators such as QEMU [3] and Rosetta2 [2] transform source binaries into different target native code, enabling users to run emulators in many different ISA. Dynamic binary translators can easily handle both the code discovery and the code location problems [55], while it introduces some runtime overhead. On the contrary, researchers have designed some static translators for higher throughput. There are also some open-source translators [44], [5], [14], [1] based on LLVM IR.

## 7. Conclusion

We designed and developed MAU, a highly parallel fuzzer that tests a smart contract on the GPU by converting the EVM bytecode to PTX assembly. We conducted extensive experiments to evaluate MAU, and the experimental results show that the throughput of MAU can reach 162.6K seeds/sec and 328.06K seeds/sec on the large and small benchmark, respectively, which exceeds the state-of-the-art tool with an 8.69X and 15.38X improvement. The exponential improvement enables MAU to detect 1.01-2.50x more bugs and explore 1.03-4.71X more code coverage than baselines.

## Acknowledgments

We thank the anonymous (meta) reviewers and shepherds for their valuable comments and instructions. This work is partly supported by Hong Kong RGC Projects (PolyU15231223 and PolyU15224121), National Key R&D Program of China (2021YFB2701000), and the Key R&D Program of Hubei Province (2023BAB017 and 2023BAB079).

## References

[1] A. Altinay, J. Nash, T. Kroes, P. Rajasekaran, D. Zhou, A. Dabrowski, D. Gens, Y. Na, S. Volckaert, C. Giuffrida, H. Bos, and M. Franz. Binrec: Dynamic binary lifting and recompilation. In *Proceedings of the 15th European Conference on Computer Systems (EuroS&P)*, New York, USA, 2020.

[2] Apple. Rosetta 2, Dec. 2021. [Online]. Available: https://developer.apple.com/documentation/apple-silicon/about-the-rosetta-translation-environment.

[3] F. Bellard. Qemu, a full-system emulation, Dec. 2021. [Online]. Available: https://github.com/qemu/qemu.

[4] G. Bertoni, J. Daemen, M. Peeters, and G. Assche. Keccak. In *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 2013.

[5] Lifting Bits. Library for lifting of x86, amd64, and aarch64 machine code to llvm bitcode, May. 2021. [Online]. Available: https://github.com/lifting-bits/remill.

[6] M. Böhme and B. Falk. Fuzzing: On the exponential cost of vulnerability discovery. In *Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, 2020.

[7] M. Böhme, V. Manès, and S. Cha. Boosting fuzzer efficiency: An information theoretic perspective. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, ESEC/FSE 2020, New York, USA, 2020.

[8] T. Chen, Z. Li, K. Fang, X. Luo, T. Wang, Y. Zhang, H. Zhu, X. Wang, H. Li, and X. Zhang. Sigrec: Automatic recovery of function signatures in smart contracts. *IEEE Trans. Softw. Eng.*, 2021.

[9] T. Chen, Y. Zhu, Z. Li, J. Chen, X. Li, X. Luo, X. Lin, and X. Zhange. Understanding ethereum via graph analysis. In *Proc. INFOCOM*, 2018.

[10] W. Chen, Z. Sun, H. Wang, X. Luo, H. Cai, and L. Wu. Wasai: Uncovering vulnerabilities in wasm smart contracts. In *Proc. ISSTA*, 2022.

[11] J. Choi, D. Kim, S. Kim, G. Grieco, A. Groce, and S. Cha. 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, 2021.

[12] Coinmarketcap. A measurement platform for cryptocurrency by market capacity., May. 2021. [Online]. Available: https://coinmarketcap.com/all/views/all/.

[13] T. David and S. Abdelrahman. Reducing branch divergence in gpu programs. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, New York, NY, USA, 2011. Association for Computing Machinery.

[14] A. Dinaburg and A. Ruef. Mcsema: Static translation of x86 instructions to llvm. In *Proceedings of the ReCon Conference*, 2014.

[15] T. Durieux, J. Ferreira, R. Abreu, and P. Cruz. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International conference on software engineering (ICSE)*, Seoul, South Korea, July, 2020.

[16] B. Falk. Vectorized emulation: Hardware accelerated taint tracking at 2 trillion instructions per second, 2018. [Online]. Available: https://gamozolabs.github.io/fuzzing/2018/10/14/vectorized_emulation.html.

[17] A. Fioraldi, D. Maier, D. Zhang, and D. Balzarotti. Libafl: A framework to build modular and reusable fuzzers. In *Proceedings of the 28th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2022.

[18] Ethereum Foundation. An high-level language for implementing smart contracts., May. 2021. [Online]. Available: https://docs.soliditylang.org/.

[19] Ethereum Foundation. Efficient gas calculation algorithm for evm., 2023. [Online]. Available: https://github.com/ethereum/evmone/blob/master/docs/efficient_gas_calculation_algorithm.md.

[20] W. Foxley. Origin protocol loses $7m in latest defi attack, May. 2021. [Online]. Available: https://www.coindesk.com/tech/2021/12/20/fantom-defi-project-grim-finance-exploited-for-30m/.

[21] A. Ghaleb and K. Pattabiraman. How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2020.

[22] Google. A distributed fuzzer based on oss-fuzzer., May. 2022. [Online]. Available: https://github.com/google/clusterfuzz/.

[23] N. Grech, L. Brent, B. Scholz, and Y. Smaragdakis. Gigahorse: thorough, declarative decompilation of smart contracts. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, QC, Canada, 2019.

[24] N. Grech, S. Lagouvardos, I. Tsatiris, and Y. Smaragdakis. Elipmoc: Advanced decompilation of ethereum smart contracts. *Proceedings of the ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications (OOPSLA)*, November, 2022.

[25] G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce. Echidna: effective, usable, and fast fuzzing for smart contracts. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 557–560, 2020.

[26] D. Havolli. Xsurge faces $5,000,000 exploit despite promises of security, Aug. 2021. [Online]. Available: https://www.bsc.news/post/xsurge-faces-5-000-000-exploit-despite-experienced-developers-and-promises-of-security.

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

[28] IBM. The redirect-on-write (row) mechanism of ibm spectrum accelerate, 2021. [Online]. Available: https://www.ibm.com/docs/en/spectrum-accelerate/11.5.4?topic=snapshots-redirect-write.

[29] Etherscan Inc. A block explorer and analytics platform for ethereum., May. 2021. [Online]. Available: https://etherscan.io/.

[30] B. Insider. A hacking of more than $50 million dashes hopes in the world of virtual currency, May. 2021. [Online]. Available: https://www.nytimes.com/2016/06/18/business/dealbook/hacker-may-have-removed-more-than-50-million-from-experimental-cybercurrency-project.html.

[31] Nikolay Ivanov, Chenning Li, Qiben Yan, Zhiyuan Sun, Zhichao Cao, and Xiapu Luo. Security threat mitigation for smart contracts: A comprehensive survey. *ACM Comput. Surv.*, 2023.

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

[33] O. Kharif. Hackers have walked off with about 14% of big digital currencies, May. 2018. [Online]. Available: https://www.bloomberg.com/news/articles/2018-01-18/hackers-have-walked-off-with-about-14-of-big-digital-currencies.

[34] T. Kolinko and Palkeo. Panoramix – decompiler at the heart of eveem.org., May. 2020. [Online]. Available: https://github.com/palkeo/panoramix.

[35] J. Krupp and C. Rossow. teether: Gnawing at ethereum to automatically exploit smart contracts. In *Proceedings of the 27nd USENIX Security Symposium (USENIX SEC)*, Baltimore, MD, USA, 2018.

[36] Knownsec Blockchain Lab. Knowsec blockchain lab xsurge flash loan attack analysis, Aug. 2021. [Online]. Available: https://medium.com/@Knownsec_Blockchain_Lab/knowsec-blockchain-lab-xsurge-flash-loan-attack-analysis-b57b75ce6a30.

[37] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2004.

[38] X. Li, P. Jiang, T. Chen, X. Luo, and Q. Wen. A survey on the security of blockchain systems. *Future Generation Computer Systems*, 2020.

[39] Y. Li, C. Feng, and C. Tang. A large-scale parallel fuzzing system. In *Proceedings of the 2nd International Conference on Advances in Image Processing*, ICAIP '18, page 194–197, New York, NY, USA, 2018. Association for Computing Machinery.

[40] D. Lin, Y. Zhang, H. Ren, S. Wang, B. Khailany, and T. Huang. Genfuzz: Gpu-accelerated hardware fuzzing using genetic algorithm with multiple inputs. In *Proceedings of the 61th ACM International Symposium on Design Automation Conference (DAC)*, 2023.

[41] LLVM. llvm releases, 2021. [Online]. Available: https://github.com/llvm/llvm-project.git.

[42] LLVM.org. Libfuzzer, May. 2023. [Online]. Available: https://llvm.org/docs/LibFuzzer.html.

[43] S. Malwa. Fantom defi project grim finance exploited for $30m, May. 2021. [Online]. Available: https://www.coindesk.com/tech/2021/12/20/fantom-defi-project-grim-finance-exploited-for-30m/.

[44] Microsoft. An aot translator can raise binaries to llvm ir., May. 2021. [Online]. Available: https://github.com/microsoft/llvm-mctoll.

[45] C. Miller. Fuzz by number: More data about fuzzing than you ever wanted to know. *Proceedings of the CanSecWest*, 2008.

[46] B. Mueller. Smashing ethereum smart contracts for fun and real profit. *HITB SECCONF Amsterdam*, 2018.

[47] T. Nguyen, L. Pham, J. Sun, Y. Lin, and Q. Minh. sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 778–788, 2020.

[48] CUDA Nvidia. Cuda toolkit documentation, 2021. [Online]. Available: https://docs.nvidia.com/cuda/.

[49] CUDA Nvidia. Parallel thread execution isa version 7.8, 2021. [Online]. Available: https://docs.nvidia.com/cuda/parallel-thread-execution/.

[50] M. Ortner and S. Eskandari. Smart contract sanctuary, 2023. [Online]. Available: https://github.com/tintinweb/smart-contract-sanctuary.

[51] M. Ren, Z. Yin, F. Ma, Z. Xu, Y. Jiang, C. Sun, H. Li, and Y. Cai. Empirical evaluation of smart contract testing: What is the best choice? In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2021.

[52] M. Rodler, D. Paaßen, W. Li, L. Bernhard, T. Holz, G. Karame, and L. Davi. Ef\cf: High performance smart contract fuzzing for exploit generation. In *Proceedings of the IEEE 8th European Symposium on Security and Privacy (EuroS&P)*, Los Alamitos, USA, 2023.

[53] S. Schroeder. Wallet bug freezes more than $150 million worth of ethereum, May. 2018. [Online]. Available: https://mashable.com/article/ethereum-parity-bug.

[54] C. Shou, S. Tan, and K. Sen. Ityfuzz: Snapshot-based fuzzer for smart contract. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2023.

[55] J. Smith and R. Nair. *Virtual machines - versatile platforms for systems and processes*. Elsevier, 2005.

[56] S. So, M. Lee, J. Park, H. Lee, and H. Oh. Verismart: A highly precise safety verifier for ethereum smart contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*, 2020.

[57] A. Souchet. A distributed fuzzer for kernel-mode targets running on windows., May. 2021. [Online]. Available: https://github.com/0vercl0k/wtf.

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

[59] L. Su, X. Shen, X. Du, X. Liao, X. Wang, L. Xing, and B. Liu. Evil under the sun: Understanding and discovering attacks on ethereum decentralized applications. In *Proceedings of the 32nd USENIX Security Symposium (USENIX SEC)*, 2021.

[60] D. Tarditi, S. Puri, and J. Oglesby. Accelerator: Using data parallelism to program gpus for general-purpose uses. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, New York, NY, USA, 2006. Association for Computing Machinery.

[61] X. Tian, H. Saito, and et.al. Llvm compiler implementation for explicit parallelization and simd vectorization. In *Proceedings of the 4th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, New York, USA, 2017.

[62] C. Torres, A. Iannillo, A. Gervais, and R. State. Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 103–119. IEEE, 2021.

[63] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, New York, NY, USA, 2018. Association for Computing Machinery.

[64] G. Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 2022.

[65] S. Wu, Z. Li, L. Yan, W. Chen, M. Jiang, C. Wang, X. Luo, and H. Zhou. Are we there yet? unraveling the state-of-the-art smart contract fuzzers. In *Proceedings of the 46th International Conference on Software Engineering (ICSE)*, Lisbon, Portugal, April, 2024.

[66] V. Wüstholz and M. Christakis. Harvey: A greybox fuzzer for smart contracts. In *Proc. European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020.

[67] W. Xu, S. Kashyap, C. Min, and T. Kim. Designing new operating primitives to improve fuzzing performance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[68] M. Zalewski. American fuzzy lop, a security-oriented fuzzer, May. 2021. [Online]. Available: https://github.com/google/AFL.

[69] Y. Zhang, C. Pang, S. Nagy, X. Chen, and J. Xu. Profile-guided system optimizations for accelerated greybox fuzzing. In *Proceedings of the 30th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, New York, USA, 2023.

```
pragma solidity ^0.4.18;
contract Owned{
  address public owner;
    // fake constructor
  function owned(){
    owner = msg.sender;
  }
  modifier isOwner {
    assert(msg.sender == owner);
    _;
  }
  function transferOwnership(address newOwner)
      ;
}
contract METADOLLAR is Owned {
    function STARTMETADOLLAR() {
        tokenBalanceOf[this] += _totalSupply;
    }
  /// @notice owner withdraw all
  function collect() isOwner {
    withdraw(this.balance);
    }
}
```

Figure 8: The source code at 0x24a7de87b3bd7298bbf8966fdf170c558d69ecc4.

## Appendix A. Overhead due to the Big Jump Table

We found that the jumps resolving (see § 4.1.3) only brings an approximately 8% overhead in terms of execution time.

The largest jump table that MAU creates has 2,093 destinations (0x447b8d7ef5aef428d49cd1fd8968c4a63b04c070). By default, LLC (IR-to-PTX) converts the table jump to nest jumps. To investigate the overhead, we curated two PTX applications:

A does a conditional jump once, representing MAU fully recovers the jump destinations.

B does a table jump once which contains 2,093 potential destinations. It jumps randomly.

A needs 12.29 us while B needs 13.31 us, which introduces about 8% overhead.

## Appendix B. Meta-Review

The following meta-review was prepared by the program committee for the 2024 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

### B.1. Summary

This paper develops a GPU-based smart contract fuzzer. By converting the smart contract to programs executable on GPU and exploiting GPU's advantages in parallel processing, it achieves significant improvements in fuzzing throughput and shows a better bug-finding capability, compared to the state-of-the-art.

### B.2. Scientific Contributions

- Creates a new tool to enable future science.
- Provides a valuable step forward in an established field.

### B.3. Reasons for Acceptance

1) This paper creates a new tool to enable future science. MAU is a novel smart contract fuzzer that utilizes GPU to test smart contracts in parallel, yielding a higher throughput and better performance.
2) The paper provides a valuable step forward in an established field. Smart contract security is critical and attracts many research efforts, including fuzzing-based vulnerability detection. Compared to existing smart contract fuzzers, the paper explores utilizing GPU's powerful parallel processing capability to accelerate smart contract fuzzing and achieve promising results, which is a valuable advancement.