EUNOMIA: Enabling User-Specified Fine-Grained Search in Symbolically Executing WebAssembly Binaries

Ningyu He
Key Lab on HCST (MOE), Peking University
China

Zhehao Zhao
Key Lab on HCST (MOE), Peking University
China

Jikai Wang
Huazhong University of Science and Technology
China

Yubin Hu
Beijing University of Posts and Telecommunications
China

Shengjian Guo
Baidu Research
USA

Haoyu Wang
Huazhong University of Science and Technology
China

Guangtai Liang
Huawei Cloud Computing Technologies Co., Ltd.
China

Ding Li*
Key Lab on HCST (MOE), Peking University
China

Xiangqun Chen
Key Lab on HCST (MOE), Peking University
China

Yao Guo*
Key Lab on HCST (MOE), Peking University
China

ABSTRACT

Although existing techniques have proposed automated approaches to alleviate the path explosion problem of symbolic execution, users still need to optimize symbolic execution by applying various searching strategies carefully. As existing approaches mainly support only coarse-grained global searching strategies, they cannot efficiently traverse through complex code structures. In this paper, we propose EUNOMIA, a symbolic execution technique that supports fine-grained search with local domain knowledge. EUNOMIA uses AES, a DSL that lets users specify local searching strategies for different parts of the program. EUNOMIA also isolates the context of variables for different local searching strategies, avoiding conflicts. We implement EUNOMIA for WebAssembly, which can analyze applications written in various languages. EUNOMIA is the first symbolic execution engine that supports the full features of WebAssembly. We evaluate EUNOMIA with a microbenchmark suite and six real-world applications. Our evaluation shows that EUNOMIA improves bug detection by up to three orders of magnitude. We also conduct a user study that shows the benefits of using AES. Moreover, EUNOMIA verifies six known bugs and detects two new zero-day bugs in Collections-C.

CCS CONCEPTS

• **Security and privacy** → **Software security engineering**; • **Software and its engineering** → **Software verification and validation**.

KEYWORDS

Symbolic Execution, Domain Specific Language, Path Explosion, WebAssembly

ACM Reference Format:
Ningyu He, Zhehao Zhao, Jikai Wang, Yubin Hu, Shengjian Guo, Haoyu Wang, Guangtai Liang, Ding Li, Xiangqun Chen, and Yao Guo. 2023. EUNOMIA: Enabling User-Specified Fine-Grained Search in Symbolically Executing WebAssembly Binaries. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597926.3598064

1 INTRODUCTION

Symbolic execution [37] is a technique for finding software bugs in various systems [6, 42, 60, 67, 68]. However, symbolic execution suffers from the *path explosion* problem. Researchers have proposed heuristics and machine learning models to prioritize program paths [30, 50, 51, 64, 71]. Other techniques aim to reduce the path exploration cost [6, 48, 56]. However, these techniques are not effective for programs with complex control flow, e.g., loops [4]. In practice, users often need to utilize various search strategies to guide symbolic execution for their analysis goals.

We observe that existing guiding approaches [6, 7, 13, 15, 17, 32, 47, 57, 61] are often too coarse-grained for some analysis purposes. They mostly support a global search strategy for the whole program, but different code blocks may fit different local strategies. For example, assume a nested loop parses network packets and the developer wants to check buffer overflow in it. The inner layer has a complex function that takes a lot of time to verify. The developer may want to prioritize other parts in the inner loop to maximize coverage. However, existing approaches cannot do that. They either get stuck in complex functions or generate unsound results. Hence, users need hints for *local search strategies* for different code blocks.

Towards this end, we present EUNOMIA, a symbolic execution framework that enables fine-grained search strategies for different program parts. Users can specify different prioritization strategies for different loop layers. This makes EUNOMIA much faster than global search strategies in bug finding. To support local search strategies, we face two challenges. The first is how to specify them for different program parts. To address this challenge, we propose AES, a DSL that lets users specify local searching strategies with few code lines. AES has parameterized operations that allow users to build customized search strategies to code blocks that depend on a specific variable. Users can bind the local search strategies to

ISSTA '23, July 17–21, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0221-1/23/07...$15.00
https://doi.org/10.1145/3597926.3598064

variables precisely with little manual effort. The second challenge is avoiding conflicts among local search strategies. A variable may belong to multiple code structures (e.g., shared by different loop layers). Multiple local search strategies may apply to the same variable, causing conflicts. To address this challenge, we propose an *interval-based path searching* algorithm that isolates the variable context into intervals, avoiding conflicts naturally.

Besides the above technical contributions, this paper implements Eunomia as a symbolic execution engine targeting full-feature WebAssembly (Wasm) [62] binaries. Wasm is an emerging hardware-independent language that has been widely adopted by web applications [29, 34, 58], blockchain apps [2, 44, 46], and serverless applications [22]. The existing state-of-the-art symbolic execution engine for Wasm lacks full support for Wasm Interface (WASI), causing limited application scopes. To the best of our knowledge, Eunomia is the first symbolic executor that supports the full features of Wasm binaries, which could be compiled from languages such as C/C++ and Go.

We evaluate Eunomia with a widely-used micro-benchmark suite and six real-world applications from various sources, including system utilities and well-known tools written in C and Go. In our evaluation, we show that Eunomia can reduce the execution time of symbolic execution by one to three orders of magnitude on the micro-benchmarks. In real-world applications, with the assistance of Aes scripts, Eunomia detected six known real-world bugs in less than a minute. Moreover, Eunomia has also discovered two new bugs in a real-world C library, Collections-C [55], which the developers have confirmed. In comparison, applying only global guiding strategies can reach neither of these two bugs within two hours. The results of a comprehensive user study also prove the utility and expressiveness of Aes. It only takes 3.1 minutes for students on average for composing an effective Aes script. We recruited 12 students to identify vulnerabilities 48 times in total. With Aes, the students can trigger the wanted vulnerability 47 times within 150 seconds. On the contrary, the students can only succeed 20 times with KLEE primitives.

We summarize our main contributions as follows:

- We design and implement a new symbolic execution framework, Eunomia, whose path-searching process can be tuned by user-specified domain knowledge at a fine-grained level without any modifications to the target programs.
- We propose a novel DSL, Aes, through which users can bind a set of local fitness functions to accelerate the analysis process. Moreover, users can also introduce pre- and post-conditions for statements or functions and even to-be-checked predicates on arbitrary locations.
- We propose a new path search strategy, *interval-based path searching*, which can isolate symbolic states into different contexts. To this end, states can be arbitrarily pruned and reordered without affecting the consistency of final results.
- We thoroughly evaluate Eunomia on a widely-used symbolic execution benchmark suite and several real-world applications. Moreover, we found two new vulnerabilities in in a 2.5k star GitHub project (Collections-C), which have been acknowledged and patched by the developers.
- To the best of our knowledge, Eunomia is the first symbolic execution framework that supports the full features of Wasm, while it also outperforms the current state-of-the-art symbolic execution tools in efficiency.

Availability: Eunomia is available at: https://github.com/HNYuuu/Eunomia-ISSTA23.

2 MOTIVATING EXAMPLE

We use Listing 1 as a motivating example for our approach, inspired by real-world industrial network protocols [5, 18]. It contains a function, `check_sections`, which takes a section vector (`sec_vec`), and the number of fields of each vector (`sec_field_cnt`). Each section has at most five fields: `token`, `index`, and `checksum` are metadata; `len` indicates the length of `data`, whose correctness is validated by `checksum`. The integers in `sec_field_cnt` indicate whether the section contains data or not. The function `check_sections` validates the fields of all sections in a two-layer nested loop. The outer loop iterates all received sections, while the inner one iterates all fields and conducts the corresponding validation.

```
 1 #define DATA(x) *((int32_t*)(x))
 2 #define sec_cnt 16
 3
 4 struct sec {
 5   int32_t token,
 6   int32_t index,
 7   int32_t checksum,
 8   int32_t len,
 9   char*   data
10 };
11
12 enum sec_name {
13   TKN, IDX, CSUM, LEN, DATA
14 };
15
16 void check_sections(sec* sec_vec, int* sec_field_cnt) {
17   int32_t i, j;
18
19   for (i = 0; i < sec_cnt; i++) {
20     sec* crt_sec = &(sec_vec[i]);
21     // Iterate each field in section
22     for (j = 0; j < sec_field_cnt[i]; j++) {
23       if(j == TKN) {
24         char* token = token_hash(DATA(crt_sec + j*4));
25         if (isValid(token)) {
26             foo();  // Expensive computation
27         } else {
28             bar();
29         }
30       } else if (j == IDX) {
31         int32_t index = DATA(crt_sec + j*4);
32         // Determine the correponding slot for data
33       } else if (j == CSUM) {
34         int32_t check_sum = DATA(crt_sec + j*4);
35         assert (check_sum & 0xabcddcba) == 0x10000;
36       } else if (j == LEN) {
37         int32_t lenth = DATA(crt_sec + j*4);
38         assert (length < 65520 && length >= 0);
39         // Allocate memory for data
40       } else {
41         char* data = curent_sec + j*4;
42         // Store data into the allocated memory
43       }
44     }
45   }
46 }
```

Listing 1: Example code for parsing received packets

Directly running symbolic execution on Listing 1 cannot complete the verification within a reasonable amount of time due to path explosion. In practice, developers could provide two pieces of *domain knowledge* to accelerate the analysis, shown as follows:

DK1 Prioritize the less-expensive *else* branch and postpone the analysis of the expensive function `foo` while verifying the user token (L23 – L29).

DK2 To avoid getting stuck in the analysis of the complex *data* field (L41), the symbolic execution can finish analyzing the simple fields firstly, i.e., *token*, *index*, and *checksum*.

Limitations of Existing Tools. Unfortunately, existing tools have no effective way to utilize **DK1** and **DK2**. We take KLEE, one of the most popular symbolic execution engines as the representative to demonstrate the limitations. Other popular tools, such as CBMC [39], also suffer from similar limitations.

KLEE cannot effectively apply **DK1** and **DK2** because it has no primitives for prioritization. Typically, we use KLEE primitives, like `klee_assume(cond)` and `klee_prefer_cex(obj, cond)` for specifying extra constraints in symbolic execution. Unfortunately, those primitives can only prune unwanted states instead of prioritizing interesting paths. Specifically, `klee_assume(cond)` can be used to insert extra constraints, and the paths that do not meet `cond` will be pruned. As for `klee_prefer_cex(obj, cond)`, it adds a preference of values for symbolic parameters of the function to be tested. It can be only used in the test driver instead of in any places in the code.

For **DK1**, the closest approximation that KLEE can make is to add `klee_assume(isValid(token)==0)`, which prunes away the branches that contain `foo()`. However, In **DK1**, we only want to prioritize the branches that lead to `bar()`. Directly pruning away paths may undermine the soundness of the analysis. Similarly, KLEE cannot utilize **DK2** either. The closest approximation is to add `klee_assume(j<3)` right after L22. However, this approach also compromises the soundness of the analysis since KLEE directly drops the analysis for `LEN` and `DATA` fields.

In summary, existing tools like KLEE and CBMC lack a flexible mechanism to prioritize the execution of certain feasible paths, so they have limited capability to improve the execution performance by utilizing rich domain knowledge from the users.

Note that although there are other work that prioritize execution paths [30, 31, 43, 50, 65], they cannot properly utilize user-defined domain knowledge as well. Existing path prioritization approaches either rely on pre-defined heuristics, black-box strategies, or even machine learning algorithms. Their goal is to accelerate symbolic execution in general instead of adopting user-defined domain knowledge. Thus, they are mostly orthogonal to our work.

Our Solution. In this section, we provide a sample code of Aes that utilizes **DK1** and **DK2** for Listing 1. We will discuss the formal definition of Aes in §3.1.

```
1 checker : func(check_sections) {
2   // DK1: prioritize bar()
3   call(bar) {prior = HIGHER;}
4   // DK2: prioritize verifications on metadata fields
5   puse(sec_field_cnt[i]) and puse(j) {
6     prior = HIGHER if j < CSUM else LOWER;
7   }
8 }
```

Listing 2: The Aes script for the guidance in §2

The 8-LOC Aes script in Listing 2 formalizes the DKs raised in §2. Two statements at L3 and L5 interpret the knowledge of **DK1** and **DK2**, respectively. Specifically, each statement is composed of two parts, i.e., the *localization part* and the *knowledge part*. We can see that these two statements are wrapped in a `checker` that works for a function `check_sections` (L1). As for the **DK1**, the localization part indicates the knowledge will be attached to the position where the function `bar()` is invoked. And in the knowledge part, we can set this branch with a higher priority than the *if* branch that calls `foo`. To this end, Eunomia will first execute L28 in Listing 1 rather than exploring both L26 and L28.

Aes handles knowledge **DK2** at L5 and L6. L5 has two `puse` expressions (ref. Figure 2) that localize the interested program point. In this case, `puse(sec_field_cnt[i])` and `puse(j)` refer to the location where both `sec_field_cnt[i]` and `j` are used as branch *predicates* in the testing program. As a result, the knowledge at L5 will be attached to the inner loop, i.e., L22 in Listing 1. Then, under the context of L5, if `j` is less than `CSUM`, i.e., verifying the first three fields, we set those three branches with `HIGHER` priority. Otherwise, e.g., for the symbolic states that verify `LEN` and `DATA`, we will set the priority as `LOWER`. In other words, the enforced behavior by Listing 2 is: (1) verify the first three metadata fields; (2) jump to the inner loop condition checking without verifying *length* and *data*; (3) move to the next section and repeat (1) & (2); and (4) deal with the remaining *length* and *data* fields once all first three steps finish.

3 DESIGN OF EUNOMIA

The workflow of Eunomia is presented in Fig.1. Eunomia takes the source code of the to-be-analyzed program and an Aes script as input. The CFG of the given program will be partitioned into *intervals* (detailed in §3.2.1), where each of them can be regarded as an independent context. Based on intervals, we propose an interval-based path searching algorithm. The algorithm maintains a priority queue for states whose priority scores are evaluated by local fitness functions that are provided in Aes scripts. To this end, the algorithm pops out the state with the highest score and one of its following basic blocks as input for the instruction simulator. The simulator conducts symbolic execution on the state according to instructions in the basic block and returns one or multiple states if path forking is necessary. Note that states will be evaluated under their corresponding contexts. Such an iteration continues until no candidate states are in the queue or the analysis is terminated. Eunomia will finally output all satisfiable paths.

In the rest of this section, we will discuss technical details of Aes and the Eunomia Execution Engine.

3.1 Auxiliary Eunomia Script

Eunomia aims to help users provide fine-grained domain knowledge to accelerate the symbolic execution process. To this end, it provides a DSL named Auxiliary Eunomia Script (Aes), which allows users to specify local search strategies with a few lines of code. Through Aes, users can bind customized prioritization functions and extra constraints on states related to specific variables.

The critical challenge of Aes is to provide an effective way for users to locate where to prioritize (or de-prioritize) during symbolic execution. One straightforward method is to let users specify line numbers or functions that should be prioritized. However, this method has two limitations. First, it will be tedious when the user wants to specify multiple lines that follow the same pattern. Second, specifying line numbers is too coarse-grained. Since symbolic execution propagates data and control flow dependencies of program states on the granularity of variables, a coarse-grained strategy

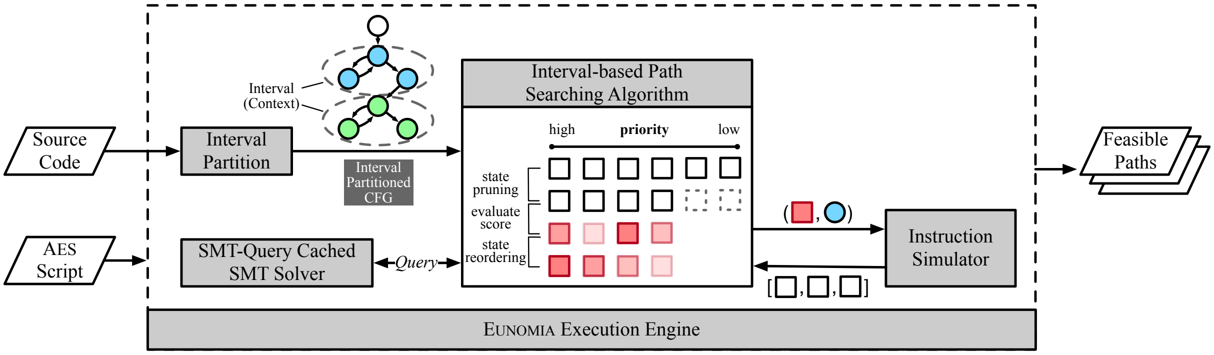

Figure 1: The workflow of Eunomia. Symbols □ and ○ refer to engine states and basic blocks, respectively.

Part I			
	bop	::=	and | or | > | < | = | ≥ | ≤ | ≠
		|	+ | − | × | / | %
	uop	::=	not | $
	l	::=	integer literal | float literal | char literal
		|	string literal | **true** | **false**
	id	::=	**halt** | **cons** | **prior** | identifier
	e	::=	l | (e) | e_1 *bop* e_2 | *uop e*
		|	identifier | identifier[identifier]
	s	::=	e | $s_1; s_2$ | $id := e$
		|	s_1 **if** e **else** s_2 | **while** e **do** s
Part II			
	locT	::=	luse | puse | cuse | ouse | ause
		|	def | func | call
	locE	::=	*locT* (identifier) | *locT* (l) | *locT* (*bop*)
		|	*locE* and *locE* | *locE* or *locE* | not *locE*
Part III			
	var	::=	$id := e$
	behave	::=	*locE* $\{s_1; \ldots; s_n\}$
		|	pre *locE* $\{s_1; \ldots; s_n\}$
		|	post *locE* $\{s_1; \ldots; s_n\}$
	advice	::=	*var* | *behave*
	pilot	::=	identifier $\{advice_1; \ldots; advice_n\}$
		|	identifier : *locE* $\{advice_1; \ldots; advice_n\}$

Figure 2: The syntax of Aes.

cannot precisely locate a variable when a line of code has multiple variables and introduces ambiguity.

To avoid those limitations, we propose a more intuitive method, allowing users to bind local search strategies to variables based on their names and usage patterns. This idea partly refers to the classical *def-use* [49] in data flow testing. We will formally discuss the syntax and semantics of Aes, and give a concrete example of Aes script targeting the problem in Listing 1.

3.1.1 *Syntax & Semantics of* Aes.

Fig. 2 gives the syntax of Aes, which is divided into three parts according to their functionalities. We will explain their semantics, respectively, in the following.

The terms in Part I are basic operators for specifying local guiding methods, like binary operator (*bop*), unitary operator (*uop*), literal (*l*), identifier (*id*), expression (*e*), and statement (*s*). Two points should be noted. First, the $ in *uop* is used to extract operands. For example, $0 corresponds to the first operand of an operator. Second, three identifiers are reserved, i.e., `halt`, `cons`, and `prior`, each of which should be followed by an expression. Through these reserved identifiers, users can formalize the corresponding domain knowledge. Specifically, if `halt` is set to `true`, it means that the user intends to terminate the whole analysis. The expression that follows `cons` can be regarded as a predicate that the user wants to verify. Finally, expressions after `prior` can be regarded as fitness functions, according to which the priority of states can be evaluated. A concrete example of accelerating the symbolic execution process according to these three variables is illustrated in §2.

Part II contains the keywords that conduct localization via functionalities of variables. Specifically, we design eight def-use relations in Aes, which are listed in *locT*. The `luse`, `puse`, `cuse`, `ouse`, and `ause` refer to the *location use*, *predicate use*, *calculation use*, *output use*, and *argument use*, respectively. Moreover, the `def`, `func`, and `call` correspond to *variable definition or assignment*, *function definition*, and *function invocation*, respectively. By applying *locT* on literals, identifiers, and binary operators, users can specify locations precisely (*locE*). Furthermore, *locE* can be combined by logical operators to limit the scope. For example, if a user intends to filter out all the "+" operators in the function `foo` and `bar` to verify if there are integer overflows, `(func(foo) or func(bar)) and call(+)` can meet his expectation.

Part III contains the keywords that combine the formalized domain knowledge (declared in Part I) and its corresponding positions (declared in Part II). The core term in Part III is *pilot*. An Aes script consists of one or multiple *pilot*. Each *pilot* can be bound on a specific position by *locE* to narrow down the adopted scope, typically a function like `func(foo)`. Within a *pilot*, users can propose concrete *advice*. Two kinds of *advice* exist: defining auxiliary variables by *var*, or declaring concrete behaviors that should be performed on specific positions by *behave*. Note that, a *behave* could be further modified by two keywords: `pre` and `post`, which hint the engine the check process should be performed *before* of *after* the bound position, respectively. For example, the *behave*: `post call(foo)` `{cons = (i > 5);}`, will additionally check if the `i` greater than 5 *after* the invocation of the function `foo`. Semantically, `pre` and `post` are only valid for those *behave* with `cons` defined in.

3.2 Eunomia Execution Engine

A critical challenge to realizing the local search strategies is isolating the context of Aes variables. For example, the DK3 in Listing 2 intends to guide the execution of the inner loop, but the engine cannot effectively distinguish which loop the `prior` at L10 refers to. To eliminate the ambiguity, we propose an *interval-based* method that partitions a CFG into orthogonal sub-graphs, automatically isolating the variable context by sub-graphs. In the rest of this section, we explain the definition of the interval, and propose an interval-based path searching algorithm, in which states can be pruned by user-added constraints or reordered by fitness functions.

3.2.1 *Interval.*

Intuitively, an interval is a sub-graph of CFG that contains no more than one loop. Formally, given a graph G consisting of nodes b_i, we can define *closed path* as $(b_1, \ldots, b_n)$ where edges of each adjacent pair of nodes (b_i, b_{i+1}) exist in G and $b_1 = b_n$. Once designating a node h as a header, interval $I(h)$ is defined as *the maximal, single entry sub-graph for which h is the entry node and in which all closed paths contain h* [1].

Given a CFG G, we can partition it into intervals with an iterative method [1]. The high level procedure of the algorithm is as follows:

1. Initiate a queue H for header nodes, and append the entry of G into H as b_0;
2. Pop the leftmost element from H, say h, and build interval $I(h)$. Specifically, for any node b in G, if all immediate predecessors of b are already in the $I(h)$, the b should also be appended into the $I(h)$. The construction of the $I(h)$ will terminate if no b meets the condition;
3. If some of (not all of) first appeared predecessors of a node b are in an interval already, insert the b to H;
4. Pop the leftmost element from H, and repeat step 2 to 4 till all nodes are partitioned into intervals.

Take the CFG of a nested loop shown in Fig. 3 as an instance to illustrate the construction process of intervals, starting from the entry, node 1. A new interval is initiated and takes node 1 as its header, dubbed as $I(1)$. Then, $I(1)$ tries to absorb node 2 into it. However, since an immediate predecessor (node 6) of node 2 is not in any known intervals, a new interval should be initiated that takes it as the header ($I(2)$). As all immediate predecessors of node 3 and node 4 are in $I(2)$ already, both of them can be included in the $I(2)$. Similarly, node 5 will be taken as a new header, where $I(5)$ consists of nodes 5, 6, and 7. Consequently, the CFG is partitioned by three intervals, i.e., $I(1)$, $I(2)$, and $I(5)$, whose topological relationship is sequential. Note that, we keep the relation from node 6 to node 2, i.e., from $I(5)$ to $I(2)$, which is not an actual edge. Thus loops can be partitioned into independent intervals though for a nested loop.

3.2.2 *Interval-Based Path Searching Algorithm.*

We propose a new algorithm, named *interval-based path searching algorithm* (see Algorithm 1), which can conduct symbolic execution inter- and intra-intervally. The algorithm takes an empty engine state and the basic entry block of the given CFG as inputs. and returns states corresponding to all feasible paths as outputs. Generally speaking, the core of the algorithm is an *iteration*. On the one side, a priority queue maintains all states with their corresponding scores calculated by the fitness functions following `prior` and dispatches the one with the highest score with its successive basic block to the engine. On the other side, the engine symbolically executes the given basic block, updates the state, and performs necessary forking on states which are appended into the priority queue. Note that states may jump over different intervals, corresponding to different Aes's contexts. Thus context switching and restoring should be performed in the implementation of the algorithm

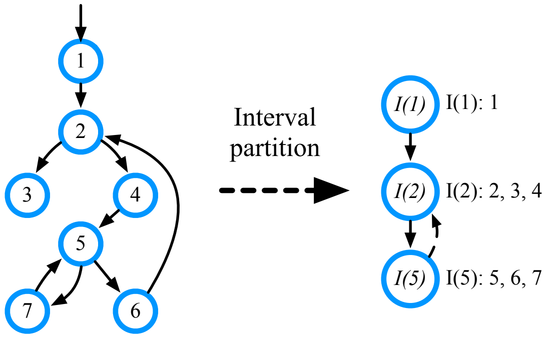

Figure 3: A CFG and its corresponding partitioned intervals.

Delving deeper into the algorithm, the `main` function at L3 schedules all states according to their priority scores through maintaining a data structure called *meta state*, dubbed as *ms* at L4, which packs the engine returned state (*es*), current basic block and its predecessor (*cur* and *pre*), all visited intervals' head (*trace*), and a mapping from localization expressions to the corresponding *advice* declared in *pilot* (*vars*). At L6, a priority score of the initiated *ms* will be assigned a default value, which is neither the highest nor the lowest value. The queue will pop out the meta state with the highest priority and pass it to `traverse` to perform symbolic execution. All feasible returned engine states will be dumped finally as outputs.

The goal of `traverse` is updating interval context and putting newly generated meta states into the priority queue. Specifically, it first collects all forked engine states (L15), and examines if possible subsequent paths exist (L16). If no subsequent path is obtained, indicating the current path is analyzed thoroughly, all engine states will be returned (L17). Otherwise, to avoid unnecessary exploration, it will screen out all unsatisfiable states according to the satisfiability of collected path conditions and the predicates given by the `cons` (L19). Then, from L21 to L29, the contexts will be updated, including visited intervals and values declared in *vars*. Finally, a new meta state and its newly evaluated priority score will be packed and appended into the priority queue (L30).

From L21 to L30, we can sum up that the algorithm adopts a BFS-like strategy for the priority scheduling and coverage of engine states, and a DFS-like strategy for switching and restoring contexts. If no Aes script is provided, the priority queue can be considered a regular FIFO queue. Thus, the algorithm is equivalent to BFS, which guarantees the correctness of our algorithm. Taking advantage of the characteristics of the algorithm, users can customize a fitness function in Aes script to dig a loop deeper without influencing other loops' recursive time and the correctness of execution.

Algorithm 1: Interval-based path searching algorithm.

```
Input  : init_es: initial engine state;
         entry_blk: entry block.
Output : res: a list of possible states in accordance with all
         feasible paths of a function.
1  heads ← calc_interval_heads();
2  q ← priority_queue();
3  Procedure main(init_es, entry_blk)
4      ms ← (es = init_es, cur = entry_blk, pre = None, trace =
         stack(), vars = map());
5      res ← [];
6      q.put(pack(priority=ms.vars[prior], item=ms));
7      while ms ← q.pop_highest() do
8          halt, ess ← traverse(ms.priority, ms.item);
9          if halt then break;
10         res.extend(ess);
11     end
12     return res;
13 end
14 Procedure traverse(p, ms)          /* Traverse CFG */
15     ess ← symbolic_execute(ms.es, ms.cur);
16     succs ← succs(ms.cur);
17     if succs = ∅ then return ms.vars[halt], ess; /* End of
         one path */
18     $T_{all}$ ← ms × ess × succs;
19     $T_{avail}$ ← {t | t.es.cons ∧ ms.vars[cons], ∀t ∈ $T_{all}$} ;
20     for nms, es, succ ← iter($T_{avail}$) do
21         nms.es, nms.cur, nms.pre ← es, succ, ms.cur;
22         if heads[nms.cur] ≠ heads[nms.pre] then
23             if heads[nms.cur] ∈ nms.trace then
24                 nms.trace, nms.vars ← restore previous
                     trace and vars;
25             else
26                 nms.trace.append(heads[ms.cur]);
27                 nms ← store current vars and init a new one;
28             end
29         nms.vars ← update vars according to AES file;
30         q.put(pack(priority=nms.vars[prior],
             item=nms));
31     end
32     return false,[];
33 end
```

3.2.3 State Scheduling. Since the algorithm maintains states in a queue, states are independent of each other, i.e., they can be arbitrarily ordered. Moreover, taking advantage of the characteristics of intervals, each state can possess its fitness functions (provided by the `prior` in AES) or constraints (provided by the `cons` in AES) under a non-global context. The highlighted four lines in Algorithm 1 illustrate how states are pruned and reordered by domain knowledge provided by users. We will detail these two processes in the following.

Pruning Unsatisfiable States. The unsatisfiable engine states returned by the engine will be pruned as soon as possible to improve the performance. Except for path conditions that are collected during symbolic execution, there are also predicates provided by users through `cons`. For instance, if a user knows the precondition of a function `bar`, that is, its argument `arg` should always be smaller than 256. The user can bind the `arg` and give a piece of advice like:

```
pre call(bar) and ause(arg) {cons = (arg < 256)}
```

State pruning will be achieved by adding the given predicates on path conditions, and verifying satisfiability by querying the backend SMT solver.

The process of state pruning is shown at L19 in Algorithm 1. After symbolically executing an engine state, one or several engine states are collected (L15). Also, the algorithm extracts all possible successor basic blocks as candidate blocks (L16). Except for reaching the end of a path, all possible paths will be enumerated by a Cartesian product (L18), where T_{all} can be represented as:

$$T_{all} = \{(ms, es_i, succ_j)\}, 0 \le i \le |ess|, 0 \le j \le |succs|$$

Only the tuples, whose both engine state's constraints ($t.es.cons$) and predicates defined by AES ($ms.vars$[`cons`]) are satisfiable, will be kept in T_{avail}.

Except for pruning unsatisfiable states, the algorithm will also consider the state assigned `halt = true` by users. At L17, once the symbolic execution reaches the end of a path, i.e., no successive basic blocks, the engine state with its value of `halt` will be returned to the function `main`. If the `halt` is set, the analysis will terminate immediately, which is often used in verifying the satisfiability of a property eagerly. Note that such a customized termination on arbitrary positions does not require any modification of the running program and the framework.

Reordering States. Once users provide `prior` via an AES script, the Scheduler can sort these states according to calculated scores each round in descending order. In the existing symbolic execution engine, dynamic state reordering by introducing human knowledge is impractical. Either extensive modification is required to modify the engine's path search strategy to eliminate the dependency between states, or the score of states is controlled by black-box machine learning algorithms. As mentioned in §3.2.1, the given CFG is partitioned by independent intervals. To this end, we can arbitrarily pick a state and run it as long as the context is changed or restored to the corresponding interval.

The implementations of state reordering locate at L6 and L30 in Algorithm 1. At L6, a score will be evaluated by the *advice* bound on the current interval once a meta state is initiated. Additionally, the loop at L20 will traverse all feasible states derived from the above pruning step. Depending on whether the to-be-traversed interval has been accessed (L23), the context of the interval would be restored or initiated. Whichever of the two scenarios occurs, the *var* in the interval should be updated (L29). Thus, L30 will recalculate a new score for the current interval. Because the 2-tuple: meta state and its score, will be packed and appended into the priority queue, all the states will be reordered, and the highest one will be picked out each time at L6.

3.3 Implementation and Optimization

We choose the WebAssembly [62] (Wasm) as the target language for Eunomia since it is emerging in several critical areas, including web applications [29, 34, 58], blockchain apps [2, 44, 46], and serverless applications [22]. The current state-of-the-art symbolic execution engine is a commercial open-source tool, Manticore [45], which requires substantial manual efforts to model the APIs of Wasm runtime to analyze Wasm applications. To ease the burden of security researchers in analyzing Wasm binaries, we implement Eunomia as the FIRST symbolic execution engine that has full support for commercial off-the-shelf Wasm applications with about 8K Python3 code. Moreover, to ensure the efficiency of Eunomia, we propose several optimizations specified to Wasm binaries, which will be detailed in the following.

3.3.1 Memory Modeling. WebAssembly adopts linear memory as the memory model. Data in its memory is raw bit string and can be indexed and interpreted. To emulate load and store via a concrete pointer, we adopt the mapping structure proposed by [32], where the value is a raw bit string modeled by BitVector, and the key is its corresponding address range. However, this model does not correctly deal with symbolic pointers.

To address the symbolic pointer problem [37], we adopt the *fully symbolic memory* model [4]. Specifically, if the loaded address is a symbol, Eunomia considers all its possible positions. Instead of forking multiple states as KLEE [7] does, which introduces enormous overhead, we transfer the burden to the SMT solver as it constantly updates on solving such constraints [7, 8, 19, 52]. In other words, we utilize *if-then-else (ite)* statements to enumerate all possible positions. For example, we need 4 bytes loaded from symbolic address ptr_a, where the current memory is $\{(0, 5) \mapsto v\}$. By an `ite` statement, we finally load:

$$\texttt{ite}(ptr_a = 0, v[0:4], \texttt{ite}(ptr_a = 1, v[1:5], \texttt{invalid}))$$

, where all possible addresses are iteratively taken, and the corresponding data is extracted from the BitVector v. If ptr_a cannot be any of the valid addresses, a symbol, `invalid`, will be returned to indicate the end of the path. As for storing data through symbolic pointers, it works similarly. An `ite` would enumerate all feasible positions to insert the data and update the corresponding value.

3.3.2 External Functions Emulating. A Wasm binary is dedicated to running in a virtual environment, which plays as an intermediary between the binary and an operating system. To this end, the engine should consider the *external environment problem* [4]. In the engine, we apply summary-based techniques to handle this problem. Specifically, there is a *WebAssembly Interface* (WASI) [63], which defines a standard interface for Wasm binaries to interact with the external environment. WASI mainly comprises IO-related functions, like `fd_write` and `fd_open`. To this end, we referred the documentation and modeled all these IO-related functions to emulate the response from the external environment. Moreover, we also summarize behaviors of common standard library functions in C and Go, including arithmetic operations, and string and memory manipulating functions. Consequently, all the invocations to the external will be intercepted. The corresponding fields in each state will be updated according to the function summary.

3.3.3 SMT-Query Cache. Determining the satisfiability of collected constraints is a challenging problem, which is time- and resource-consuming [16]. Therefore, we have designed a cache pool for querying to alleviate this problem. Formally, we define the SMT-query cache as a set $C = [c_1, c_2, ..., c_n]$ that contains all solved constraints. For each $c_i \in C$, our cache pool caches its result and all lemmas inferred from it. Then, for a given constraint c_{solve} that needs to be solved, before asking SMT solvers for solving, Eunomia first queries the cache C following three rules:

- If $c_{solve} \in C$, Eunomia directly returns the result.
- If $c_{solve} \notin C \land \exists c_i \in C, c_i \subset c_{solve} \land c_i = UNSAT$, Eunomia sets c_{solve} as UNSAT.
- If $c_{solve} \notin C \land \exists c_i, c_j \in C$, where $c_i, c_j \subset c_{solve}, |c_i| \geq |c_j|$, c_i is the cached *maximal subset* of c_{solve}. Eunomia first initializes the SMT solver's solving context with c_i and the cached lemmas inferred from c_i. Then, it adds the constraint $c_{solve} - ci$ to the SMT solver for incremental solving and avoids calculating the results of c_i again.

If c_{solve} does not match all three rules, Eunomia send c_{solve} to the SMT solver and cache the result.

4 EVALUATION

We aim to evaluate the efficiency and the effectiveness of Eunomia. Specifically, we answer the following research questions:

RQ1 Is Eunomia more efficient than state-of-the-art tools?
RQ2 Is Eunomia also more effective for bug detection?
RQ3 Is Aes easy-to-use for non-expert users?

4.1 Benchmark

We evaluate Eunomia on both micro-benchmark programs and real-world applications. Logic Bomb [66] is the used micro-benchmark that has 64 test cases for evaluating the performance of symbolic execution tools from 12 aspects like symbolic memory, external functions calls, floating numbers, and so on.

Our real-world application set contains six open-source applications/libraries. The first three are actively maintained programs that have 1.4k-16k lines of C code: (1) Collections-C [55] is a common data structures library written in C; (2) DNSTracer [41] is a tool in Linux Kernel that determines where a Domain Name Server gets its information from for a given hostname; (3) GOCR [35] is an open-sourced OCR program. The rest three are Go projects: (4) Snappy [23] is a compression tool that has 6.5K lines of code and more than 1.2K stars on Github; (5) Go Image [24] is an official image manipulation library; and (6) Sprintf [25] is the official implementation of `sprintf` function in Go.

4.2 Experiment Setup

Our experiments are performed on a server running Ubuntu 18.04 with 16 Intel(R) Xeon(R) Platinum 8369B CPU @ 2.70GHz and 128G RAM. We compile all targets with clang in wasi-sdk (version `14.0`) and TinyGo (version `0.21.0`). To horizontally compare the effectiveness and efficiency brought by Eunomia, we choose Manticore [45] (version `0.3.7`) as our baseline. Specifically, Manticore is the state-of-the-art symbolic execution engine for Wasm binaries. It is not only in commercial use and actively maintained but also

Table 1: Results of symbolically executing the Logic Bomb benchmark with 5 minutes timeout limitation.

	#Success	#Fail	#Timeout	#Inapplicable
	(%)	(%)	(%)	(%)
Manticore	10 (15.63%)	11 (17.19%)	30 (46.88%)	13 (20.31%)
Eunomia	29 (45.31%)	9 (14.06%)	13 (20.31%)	13 (20.31%)

Table 2: Time in triggering logic bombs or bugs with 2 hours as timeout for Manticore (T_M), Eunomia (T_E), and Eunomia with the help of Aes (T_{E-dsl}). The two bugs marked as 0-day are new bugs discovered by Eunomia.

	Vul. Type	Manticore	Eunomia	Eunomia (Aes)
Loop#1	-	~101min	148s	89s
Loop#2	-	~30min	105s	66s
Loop#3	-	~85min	117s	85s
Loop#4	-	> 2h	> 2h	> 2h
Collections-C (`6f93d5`)	Integer Overflow	> 2h	> 2h	1.5s
Collections-C (`73c468`)	Implementation Error in `reverse` of Array (0-day)	> 2h	> 2h	33s
Collections-C (`73c468`)	Implementation Error in `reverse` of Deque (0-day)	> 2h	> 2h	35s
DNSTracer (`ver.1.9`)	Buffer Overflow	~34min	85s	1.7s
GOCR (`ver.0.40`)	Integer Overflow	> 2h	~50min	26s
Snappy (`f4b10f`)	Slice Out of Range	~57min	78s	3.2s
Go Images (`72a658`)	Division by Zero	> 2h	~21min	34s
Go `sprinf` (`a2ef54`)	Integer Overflow	~25min	56s	7s

open-sourced (over 3.2K stars on GitHub). However, some additional manual efforts are necessary, or Manticore cannot directly analyze a Wasm binary. For example, Manticore does not support imported library functions. We have to set them unreachable manually. Also, it requires an additional script in Python to emulate interactions, e.g., `getchar` and `printf`, between Wasm binaries and their external environment. Last, Manticore only regards exported functions as entries. Thus we have to export the entry for symbolic execution manually. At last, we choose z3 (version `4.8.12`) as the back-end SMT solver because both of them support it.

4.3 RQ 1: Efficiency

To answer this question, we first compare the execution time of Manticore and Eunomia on all 64 test cases in the Logic Bomb benchmark and real-world applications with BFS global searching strategy. Note that, because all these targets are compiled from standard toolchains, we made a few changes to adapt Manticore, as mentioned in §4.2. For the feasible paths analyzed by both tools on each program, we compared the number and content of the paths. This was done for two reasons: first, to confirm the correctness of the interval-based path-searching algorithm by cross-comparison; and second, to ensure that our modifications for adaptation purposes did not change the semantics of the original programs.

The results of symbolically executing the Logic Bomb benchmark are shown in Table 1. As we can see, Eunomia significantly outperforms Manticore in the number of successfully triggering bomb and timeout cases. Among all 12 categories, Eunomia has a better performance in logic bombs focusing on symbolic memory, floating numbers, and external library functions. Though there are 13 timeout cases, we find that more than half of them are due to the lack of support of the file system. Currently, Eunomia can only step into those functions that read and write files, which is time-consuming. Further, the inapplicable cases are mainly due to the language features of WebAssembly. Particularly, these cases include multi-threading (6), goto statement (3), socket communication (2), and asm code (2), which are not yet supported by WebAssembly. However, some of these features have been planned in Wasm, like multi-threading in Rust to Wasm [59].

As for analyzing real-world applications, the results are shown in the third and fourth columns of Table 2. We can see that among eight applications, Manticore can only finish the analysis on three ones within two hours, while the number is five for Eunomia. Moreover, it is easy to observe that Eunomia has one to two orders of magnitude improvement in efficiency compared to Manticore.

By observing the log messages in the experiment, we believe that this improvement in efficiency can be summarized in two points. First, Eunomia adopts the memory modeling mechanism mentioned in §3.3.1. Once encountering symbolic pointers, Eunomia will construct the corresponding `ite` statements instead of forking states with different constraints, which is time- and resource-consuming. Second, once a set of constraints is asked for solving, Manticore will initiate multiple z3 instances with different random seeds to see which one could search for a feasible solution first. However, Eunomia adopts the SMT-caching mechanism as we mentioned in §3.3.3. To evaluate the improvement introduced by SMT-caching, we rerun Eunomia by disabling SMT-caching. Our experiment shows that SMT-caching can reduce the solving time by two to three orders of magnitude. We omit the detailed result in this paper due to the page limit.

RQ-1 Answer

Even with the default global path searching algorithm, Eunomia outperforms the state-of-the-art symbolic executor Manticore. We believe this is because some features, like the memory modeling algorithm and SMT-caching mechanism, are introduced and implemented in Eunomia.

4.4 RQ 2: Effectiveness

We compare the effectiveness of Manticore and Eunomia by measuring the used execution time for triggering vulnerabilities in real-world cases and four loop logic bombs[1], as shown at the third to the fifth columns in Table 2. As we mentioned in §4.3, even with

[1] These four logic bombs are the cases under the *loop* category of the Logic Bomb benchmark mentioned in Table 1.

the identical global path searching strategy, Eunomia is tens of times better compared to Manticore in terms of bug triggering, showing excellent effectiveness in bug detection. Moreover, with the help of local path searching strategies, which are provided by users Aes scripts, the bug detection time of Eunomia can be further improved for another one to three orders of magnitude. Take the first loop logic bomb as an example. It is an implementation of the Collatz conjecture, which takes an integer as input and conducts the following simple arithmetic operations on the input:

$$\texttt{collatz(x)} = \begin{cases} x/2 & \text{if } x \text{ is even} \\ 3 * x + 1 & \text{if } x \text{ is odd} \end{cases} \tag{1}$$

The bomb iteratively trigger the Collatz function, and can be simplified as:

```
 1 int logic_bomb(int x) {
 2     int loopcount = 1;
 3     int j = collatz(x);
 4     while (j != 1) {
 5         j = collatz(j);
 6         loopcount++;
 7     }
 8     if (loopcount == 25) {return BOMB;}
 9     else {return NORMAL;}
10 }
```

To trigger the bomb at L8 as soon as possible, we introduce fine-grained knowledge like `puse(j) {prior = abs(25 - loopcount);}`. To this end, the paths with higher `loopcount` will be prioritized. As a result, the prioritized paths lead to a faster and hence more effective bug detection capability than Manticore.

RQ-2 Answer

Eunomia offers more effective bug detection capability than Manticore, no matter with the global or local search strategies. When introducing users domain knowledge by providing Aes script, it can prioritize or defer the analysis on designated parts of a program to identify bugs effectively.

4.5 RQ 3: Usability

To evaluate the usability of Aes, we have compared Aes in Eunomia with the primitives in KLEE, such as `klee_assume` and `klee_prefer_cex`. We achieve this by conducting a user study.

Specifically, we invited 12 computer science graduate students that did not participate in this project to learn Aes and KLEE primitives. Then, we asked the students to test four test cases as listed in Table 2. For each student, we first provide tutorials about KLEE and Aes to them for training. Then, we hand out mini-quizzes for KLEE and Aes[2], respectively, to them to ensure that they have learnt the knowledge. Once the students have passed the mini-quizzes, we let them compose KLEE test drivers and Aes to trigger the bugs in the given test cases. For these test cases, we told the students where to be examined and which vulnerability it has[3].

In our user study, we evaluate three aspects. First, we evaluate the efforts needed to learn KLEE primitives and Aes. This is measured by the time for the students to pass the mini-quizzes (T_L). Second, we evaluate the efforts required to use KLEE primitives and Aes, measured by the time spent to compose the KLEE test driver and Aes (T_C), respectively. Third, we evaluate the effectiveness of the KLEE test drivers and Aes. This is measured by the execution time of the symbolic engine to find the wanted vulnerability (T_E). All the results are summarized in Table 3.

4.5.1 Learning and Composing Efforts. As shown in Table 3, the efforts required to learn Aes is similar to the efforts needed to learn KLEE primitives. On average, the students used 22 minutes to pass the mini-quiz about KLEE primitives while spending 25 minutes passing the mini-quiz for Aes. Although it took three minutes longer to learn Aes on average, this is a one-time overhead for each user and is acceptable.

It took 8.8 minutes for the students to create the test driver with KLEE primitives for all four cases on average. Correspondingly, the students spent 12.5 minutes to create Aes scripts for all four cases on average. Moreover, we run the Wilcoxon test over the composing time and find there is no statistical significance between KLEE and Aes (p = .052). Thus, we conclude that the efforts required to compose Aes are similar to the effort required to create KLEE test drivers.

4.5.2 Expressiveness of Aes and KLEE Primitives. In general, the students can find more vulnerabilities with Aes than KLEE primitives because of the powerful expressiveness of Aes grammars. In Table 3, for the 48 (12*4) test cases, the students have successfully triggered the known vulnerabilities in 47 cases within 150 seconds. The only exception is from S_2 where the student failed to trigger the vulnerability for GOCR. On the contrary, students can only trigger 20 vulnerabilities with KLEE primitives within the given time quota (2 hours). Specifically, none of the students can trigger the vulnerability in GOCR. This means that KLEE primitives have fundamental limitations that prohibit users from expressing effective domain knowledge.

We further investigate why students failed to trigger the vulnerabilities in GOCR. The root reason is that KLEE cannot precisely prioritize loop branches, so it will inevitably get stuck in a vulnerability-irrelevant but computation-intensive function in GOCR. The vulnerable code of GOCR is as follows:

```
1 int main(){
2     parseImage(img, &nx, &ny);
3     for(...;i < nx*ny*3;){
4         // increase values of nx and ny
5     }
6     foo(); // computation-heavy
7 }
```

Here nx and ny represent an image's width and height, respectively. This code contains an integer overflow vulnerability. In the loop body, the values of nx and ny would increase, resulting in a potential integer overflow (nx*ny) at L3. The effective domain knowledge for this case is to prioritize the execution of the loop body and avoid analyzing the expensive function `foo()`.

Unfortunately, neither `klee_assume` and `klee_prefer_cex` can precisely express the prioritization of the loop body. Instead, according to our user study, for each of 48 cases, users can utilize the domain knowledge with at most 5 lines of Aes code. We conclude

[2] The quizzes can also be accessed at our repo.

[3] Note that, if users adopt Aes to assist the analysis on a target under real-world scenarios, they do not need to know the type and location of vulnerabilities. Part of the Aes script is to examine vulnerabilities, which can be copied directly from templates. The other part can be used to guide the control flow of symbolic execution process according to their domain knowledge. See §8 for more details.

Table 3: A user study on comparing KLEE primitives and Aes in Eunomia in terms of usability and executing efficiency. The T_L, T_C, and T_E refer to the corresponding time consumed on syntax (l)earning, primitives/Aes scripts (c)omposing, and (e)xecuting till the final results, respectively, where the T refers to (t)imeout that is set as 2 hours. Moreover, the parenthesis under T_L refers to how many questions are answered correctly. Also, S_i denotes a student where i ∈[1,12].

	KLEE			Eunomia			KLEE			Eunomia			KLEE			Eunomia			KLEE			Eunomia		
	T_L	T_C	T_E	T_L	T_C	T_E	T_L	T_C	T_E	T_L	T_C	T_E	T_L	T_C	T_E	T_L	T_C	T_E	T_L	T_C	T_E	T_L	T_C	T_E
	S_1						S_2						S_3						S_4					
Loop#1	17min	5min	T	22min	1.5min	87s	23min	7min	T	27min	7min	86s	23min	4min	T	25min	2min	88s	21min	10min	T	24min	3min	90s
Collections-C (6f93d5)		0.5min	<1s		1.5min	1s		1min	<1s		3min	1.4s		1min	<1s		2min	1.3s		1min	<1s		2min	1.1s
DNSTracer (ver.1.9)		1min	14min		0.5min	2s		1min	14min		3min	1.7s		1min	15min		3min	1.4s		1min	14min		4min	1.4s
GOCR (ver.0.40)		2min	T		0.5min	26s		1min	T		3min	T		2min	T		3min	27s		3min	T		5min	26s
	S_5						S_6						S_7						S_8					
Loop#1	22min	0.5min	T	27min	4min	88s	25min	0.5min	T	26min	5min	89s	23min	10min	T	27min	4min	149s	22min	3min	T	25min	1min	86s
Collections-C (6f93d5)		0.5min	<1s		3min	1.4s		0.5min	<1s		2min	1.6s		1min	<1s		2min	1.2s		1min	<1s		2min	1.3s
DNSTracer (ver.1.9)		1min	T		3min	2s		0.5min	14min		2min	1.9s		1min	14min		3min	2.2s		2min	T		2min	1.3s
GOCR (ver.0.40)		1min	T		6min	27s		1min	T		4min	28s		3min	T		6min	31s		2min	T		3min	28s
	S_9						S_{10}						S_{11}						S_{12}					
Loop#1	20min	10min	T	23min	5min	86s	25min	2min	T	26min	5min	86s	20min	0.5min	T	26min	5min	87s	21min	3min	T	25min	3min	90s
Collections-C (6f93d5)		2min	<1s		3min	2s		1min	<1s		2min	1.3s		1min	<1s		2min	1.2s		1min	<1s		2min	2s
DNSTracer (ver.1.9)		3min	T		4min	1.8s		1min	15min		4min	1.3s		0.5min	T		2min	1.8s		1min	15min		2min	2s
GOCR (ver.0.40)		4min	T		5min	27s		3min	T		4min	28s		1min	T		3min	27s		2min	T		3min	28s

that users can better utilize domain knowledge with Aes than with KLEE primitives.

RQ-3 Answer

Compared with KLEE primitives, Aes is an easy-to-learn DSL with better usability. By spending a similar amount of composing efforts, users can utilize domain knowledge and express precise prioritization with Aes for faster symbolic execution.

5 CASE STUDY: DETECTING NEW BUGS

In our experiments, Eunomia successfully identified six known bugs in real-world applications and two new extra 0-day bugs in Collections-C, as shown in the **Vul. Type** column in Table 2. In this section, we study the two new bugs in detail.

```
1  // reverse the given deque
2  void reverse(CC_Deque *deque) {
3      size_t i, j;
4      size_t s = deque->size;
5      size_t c = deque->capacity - 1;
6      size_t first = deque->first;
7
8      // the loop condition should be: i < s / 2
9      for (i = 0, j = s - 1; i < (s - 1) / 2; i++, j--) {
10         size_t f = (first + i) & c;
11         size_t l = (first + j) & c;
12
13         void *tmp = deque->buffer[f];
14         deque->buffer[f] = deque->buffer[l];
15         deque->buffer[l] = tmp;
16     }
17 }
```

Listing 3: The code of `reverse()` in deque

We tested all 159 interface functions provided by Collections-C by specifying their pre- and post-conditions in Aes. For each interface function, if Eunomia cannot finish the analysis in 5 minutes, we check the source code and update the Aes script to determine if local search strategies can be adopted. To better explain how Aes can detect bugs, we take the bug we discovered in the `reverse()` function of the deque data structure (Listing 3), as an instance.

The `reverse()` function takes a deque as input and extracts the deque's size, capacity, and the first element in `s`, `c`, and `first`, respectively. Then, in a loop, it calculates indices of the first and the last elements that are not reversed yet. After verifying that the indices are not beyond the capacity, it will swap them and continue the loop. However, the loop condition at L9, i.e., `i < (s - 1)`, is defective. The `i` cannot refer to the former one of the central two elements if the size of the deque is even due to the *floor division*. For symbolic execution, it is hard to trigger this bug. Because triggering this bug requires constructing a deque with symbolic length first, initiating the deque with elements, invoking `reverse()` from dozens of functions, and implementing a specific checker that examines the deque after the invocation. Taking advantage of the expressiveness of Aes and local path search strategy, we can find this bug efficiently. Listing 4 illustrates the corresponding Aes script.

```
checker {
  head_reverse_cnt = 0;
  tail_reverse_cnt = 0;

  func(reverse) {
    // update the counter on swapped elements
    def(f) and cuse(i) {head_reverse_cnt =
           head_reverse_cnt + 1;}
    def(l) and cuse(j) {tail_reverse_cnt =
           tail_reverse_cnt + 1;}

    def(s) {cons = (s >= 0 and s < 65536);}
    def(c) {cons = (c == 65535);}

    // prioritize paths heading to loop body
    cuse(s) and puse(i) {prior = HIGHER if i < (s-1)/2
           else LOWER;}
  }

  post func(reverse) {cons = (head_reverse_cnt +
         tail_reverse_cnt == s);}
}
```

Listing 4: The Aes script for Listing 3

An intuition to examine the correctness of `reverse()` in deque is that: the number of swapped elements should be equal to the size of the deque if the position of the head and the tail elements can be exchanged correctly. Therefore, based on such an intuition, we compose an Aes script as shown in Listing 4. At L2 and L3, we declare two counters to track the number of swapped elements in the head and the tail of the deque, respectively. They will be incremented by one once the indices of to-be-swapped elements are updated, which can be bound by `def(f) and cuse(i)` and `def(l) and cuse(j)`. After executing `reverse()`, the sum of these two counters should be equivalent to the size of the deque, which is checked by L17.

However, without the support of the local search strategy, such a check alone may lead to false positives. That is because the loop condition of `reverse()` (L9 at Listing 3) is unbounded (as `s` is a symbol). Once encountering this condition, path forking is performed. The forked path will not only lead to the path explosion problem but also result in false positives as the path that jumps out of the loop has not completed the reverse process at all, i.e., `head_reverse_cnt + tail_reverse_cnt == s` cannot be guaranteed. Therefore, we prioritize the branch heading to the loop body by `prior = HIGHER if i < (s-1)/2 else LOWER` (the usage here is consistent with the one in §2). To this end, only paths that head to the loop body will be executed as they have higher priority. Once the reverse is complete, the path that jumps out of the loop can be executed, and the property at L17 will be verified. The bug in `reverse()` of the Array data structure is similar. These two bugs exist in the latest release, and both of them are acknowledged and patched immediately by the developer. We urge tools that adopt Collections-C as the library to pull a new release to avoid negative impacts in their production environments.

6 THREATS TO VALIDITY

External Validity: To ensure the external validity of our experiments, we select benchmarks from different independent sources. We first use Logic Bomb [66], a well-constructed third-party benchmark suite dedicated to evaluating symbolic execution approaches. Besides, we also use six real-world applications, three written in C and three written in Go. These applications are either popular open-source projects from GitHub (e.g., with more than 1,000 stars) or the official library of Go. In general, our benchmarks can represent a wide rage of applications with different types.

Internal Validity: To ensure internal validity, we carefully control the parameters and variables in our experiment. For example, we configured different engines as similar as possible, e.g., using z3 as the backend in all cases. In order to evaluate the effectiveness of local searching strategies, we have compared the performance of Eunomia with global searching strategies in detail.

7 RELATED WORK

Many symbolic execution approaches were proposed for bug hunting on various targets [3, 6, 9, 11, 12, 21, 26–28, 33, 36, 38, 43, 45, 53, 54, 60, 67–69]. For example, Kim et al. [36] have proposed HFL, combining fuzzing and symbolic execution, and found 24 previously unknown vulnerabilities in Linux kernels. He *et al.* [32] proposed a symbolic execution engine EOSafe targeting EOSIO smart contracts and identified 27 in-the-wild attacks. One of the key problems is the path explosion problem. To this end, different approaches have been proposed to reduce the searching spaces of paths[4, 6, 10, 14, 15, 20, 30, 31, 40, 48, 50, 51, 56, 64, 70, 71]. However, current heuristics, no matter designed manually or learned by machine-learning techniques, are only effective for specific tasks and require substantial human work if migrated to other tasks.

Researchers proposed techniques to guide symbolic execution with human knowledge [15, 17, 43, 47, 57]. For instance, WOODPECKER lets users specify a point of interest and guides symbolic execution to it with program slicing [15]. Guided symbolic execution uses heuristics to reach a target [43]. VerX and SAW design DSLs for domain-specific logic [17, 47]. Unlike them, Aes models general and high-level human knowledge. These approaches lack local searching strategies, which are key contributions of Eunomia.

8 DISCUSSION

Eunomia uses users' domain knowledge to improve symbolic execution. Testing large code bases needs lots of domain knowledge. Hence, we designed Aes to let developers guide the symbolic execution easily. As shown in §4.5, a CS graduate student can write Aes scripts for real applications in less than 15 minutes. Moreover, Aes scripts can enhance efficiency and find two 0-day bugs in real-world applications (§5).

Users do not need to know the exact location and type of vulnerability beforehand. We provide templates for Aes scripts that can be reused by others. For example, `post call($+) {cons = ($0 > $1 and $0 > $2);}` can detect integer overflow on all addition operators. It will take effect automatically during symbolic execution. Users can also use their domain knowledge to focus on key parts of the program and skip the insignificant parts. As Table 3 shows, Eunomia can outperform KLEE by prioritizing or avoiding certain branches.

9 CONCLUDING REMARKS

In this paper, we have proposed a symbolic execution framework Eunomia that supports fine-grained local searching strategies with user-specified knowledge. We implement Eunomia as a platform for Wasm binaries, which supports applications written in multiple mainstream languages, such as C and Go. The experimental results show that Eunomia improves the speed of discovering bugs in real applications by up to three orders of magnitude when introducing local search strategies. Besides verifying six known bugs, Eunomia has also discovered two zero-day bugs in a popular open-source project, Collection-C.

ACKNOWLEDGMENTS

We would like to thank the anonymous ISSTA reviewers for their valuable feedback of this paper. Also, we have great thanks to all 12 students who are involved in our RQ3 user study. This work was partly supported by National Key R&D Program of China (2021YFB2701000), the National Natural Science Foundation of China (62172009, 62141208), and Huawei Research Fund. Ding Li and Yao Guo are corresponding authors.

REFERENCES

[1] Frances E Allen. 1970. Control flow analysis. *ACM Sigplan Notices* 5, 7 (1970), 1–19. https://doi.org/10.1145/390013.808479

[2] altermarkive. 2022. Crypto miner on webpages. https://github.com/altermarkive/javascript-emscripten-bitcoin-miner

[3] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. 2014. Enhancing symbolic execution with veritesting. In *Proceedings of the 36th International Conference on Software Engineering*. 1083–1094. https://doi.org/10.1145/2568225.2568293

[4] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 1–39. https://doi.org/10.1145/3182657

[5] Matthew Bocci, Thomas Nadeau, Luca Martini, Samer Salam, Ali Sajassi, and Satoru Matsushima. 2014. Inter-Chassis Communication Protocol for Layer 2 Virtual Private Network (L2VPN) Provider Edge (PE) Redundancy. RFC 7275. https://doi.org/10.17487/RFC7275

[6] Fraser Brown, Deian Stefan, and Dawson Engler. 2020. Sys: a static/symbolic tool for finding good bugs in good (browser) code. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 199–216.

[7] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.. In *OSDI*, Vol. 8. 209–224.

[8] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. 2008. EXE: Automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)* 12, 2 (2008), 1–38. https://doi.org/10.1145/1455518.1455522

[9] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S Pasareanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. 2011. Symbolic execution for software testing in practice: preliminary assessment. In *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 1066–1071. https://doi.org/10.1145/1985793.1985995

[10] Marek Chalupa, Tomáš Jašek, Jakub Novák, Anna Řechtáčková, Veronika Šoková, and Jan Strejček. 2021. Symbiotic 8: Beyond Symbolic Execution:(Competition Contribution). *Tools and Algorithms for the Construction and Analysis of Systems* 12652 (2021), 453. https://doi.org/10.1007/978-3-030-72013-1_31

[11] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. 2020. SAVIOR: Towards Bug-Driven Hybrid Testing. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 1580–1596. https://doi.org/10.1109/SP40000.2020.00002

[12] Vitaly Chipounov, Vlad Georgescu, Cristian Zamfir, and George Candea. 2009. Selective symbolic execution. In *Proceedings of the 5th Workshop on Hot Topics in System Dependability (HotDep)*.

[13] Maria Christakis, Peter Müller, and Valentin Wüstholz. 2016. Guiding dynamic symbolic execution toward unverified program executions. In *Proceedings of the 38th International Conference on Software Engineering*. 144–155. https://doi.org/10.1145/2884781.2884843

[14] Christoph Csallner and Yannis Smaragdakis. 2005. Check'n'Crash: Combining static checking and testing. In *Proceedings of the 27th international conference on Software engineering*. 422–431. https://doi.org/10.1145/1062455.1062533

[15] Heming Cui, Gang Hu, Jingyue Wu, and Junfeng Yang. 2013. Verifying systems rules using rule-directed symbolic execution. *ACM SIGPLAN Notices* 48, 4 (2013), 329–342. https://doi.org/10.1145/2499368.2451152

[16] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[17] Robert Dockins, Adam Foltzer, Joe Hendrix, Brian Huffman, Dylan McNamee, and Aaron Tomb. 2016. Constructing semantic models of programs with the software analysis workbench. In *Working Conference on Verified Software: Theories, Tools, and Experiments*. Springer, 56–72. https://doi.org/10.1007/978-3-319-48869-1_5

[18] Dennis Dube and Jacques Camerini. 2002. *MODBUS Application Protocol*. Internet-Draft draft-dube-modbus-applproto-00. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-dube-modbus-applproto/00/ Work in Progress.

[19] Bassem Elkarablieh, Patrice Godefroid, and Michael Y Levin. 2009. Precise pointer reasoning for dynamic test generation. In *Proceedings of the eighteenth international symposium on Software testing and analysis*. 129–140. https://doi.org/10.1145/1572272.1572288

[20] Dawson Engler and Daniel Dunbar. 2007. Under-constrained execution: making automatic code destruction easy and scalable. In *Proceedings of the 2007 international symposium on Software testing and analysis*. 1–4. https://doi.org/10.1145/1273463.1273464

[21] Xiang Fu and Kai Qian. 2008. SAFELI: SQL injection scanner using symbolic execution. In *Proceedings of the 2008 workshop on Testing, analysis, and verification of web services and applications*. 34–39. https://doi.org/10.1145/1390832.1390838

[22] Phani Kishore Gadepalli, Sean McBride, Gregor Peach, Ludmila Cherkasova, and Gabriel Parmer. 2020. Sledge: a serverless-first, light-weight wasm runtime for the edge. In *Proceedings of the 21st International Middleware Conference*. 265–279. https://doi.org/10.1145/3423211.3425680

[23] golang. 2022. GitHub page of Snappy. https://github.com/golang/snappy

[24] golang. 2022. Home page of image package. https://pkg.go.dev/image

[25] golang. 2022. Implementation of sprintf in Go. https://github.com/golang/go/blob/master/src/fmt/print.go

[26] Shengjian Guo, Yueqi Chen, Peng Li, Yueqiang Cheng, Huibo Wang, Meng Wu, and Zhiqiang Zuo. 2020. SpecuSym: speculative symbolic execution for cache timing leak detection. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*. ACM, 1235–1247.

[27] Shengjian Guo, Yueqi Chen, Jiyong Yu, Meng Wu, Zhiqiang Zuo, Peng Li, Yueqiang Cheng, and Huibo Wang. 2020. Exposing cache timing side-channel leaks through out-of-order symbolic execution. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 147:1–147:32.

[28] Shengjian Guo, Meng Wu, and Chao Wang. 2018. Adversarial symbolic execution for detecting concurrency-related cache timing leaks. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. ACM, 377–388.

[29] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 185–200. https://doi.org/10.1145/3062341.3062363

[30] Jingxuan He, Mislav Balunović, Nodar Ambroladze, Petar Tsankov, and Martin Vechev. 2019. Learning to fuzz from symbolic execution with application to smart contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 531–548. https://doi.org/10.1145/3319535.3363230

[31] Jingxuan He, Gishor Sivanrupan, Petar Tsankov, and Martin Vechev. 2021. Learning to Explore Paths for Symbolic Execution. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2526–2540. https://doi.org/10.1145/3460120.3484813

[32] Ningyu He, Ruiyi Zhang, Haoyu Wang, Lei Wu, Xiapu Luo, Yao Guo, Ting Yu, and Xuxian Jiang. 2021. {EOSAFE}: Security Analysis of {EOSIO} Smart Contracts. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*.

[33] Grant Hernandez, Farhaan Fowze, Dave Tian, Tuba Yavuz, and Kevin RB Butler. 2017. Firmusb: Vetting usb device firmware using domain informed symbolic execution. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2245–2262. https://doi.org/10.1145/3133956.3134050

[34] Hexops. 2022. Game engine and graphics toolkit. https://github.com/hexops/mach

[35] Joerg Schulenburg. 2022. Official page of GOCR. https://jocr.sourceforge.net/

[36] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. 2020. HFL: Hybrid Fuzzing on the Linux Kernel.. In *NDSS*. https://dx.doi.org/10.14722/ndss.2020.24018

[37] James C King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394. https://doi.org/10.1145/360248.360252

[38] Ronny Ko, James Mickens, Blake Loring, and Ravi Netravali. 2021. Oblique: Accelerating Page Loads Using Symbolic Execution. In *18th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 21)*. 289–302.

[39] Daniel Kroening and Michael Tautschnig. 2014. CBMC – C Bounded Model Checker. In *Tools and Algorithms for the Construction and Analysis of Systems*, Erika Ábrahám and Klaus Havelund (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 389–391.

[40] You Li, Zhendong Su, Linzhang Wang, and Xuandong Li. 2013. Steering symbolic execution to less traveled paths. *ACM SigPlan Notices* 48, 10 (2013), 19–32. https://doi.org/10.1145/2544173.2509553

[41] Linux. 2022. Official page of dnstracer. https://linux.die.net/man/8/dnstracer

[42] Lannan Luo, Qiang Zeng, Bokai Yang, Fei Zuo, and Junzhe Wang. 2021. Westworld: Fuzzing-Assisted Remote Dynamic Symbolic Execution of Smart Apps on IoT Cloud Platforms. In *Annual Computer Security Applications Conference*. 982–995. https://doi.org/10.1145/3485832.3488022

[43] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S Foster, and Michael Hicks. 2011. Directed symbolic execution. In *International Static Analysis Symposium*. Springer, 95–111. https://doi.org/10.1007/978-3-642-23702-7_11

[44] Mihai Maganu. 2022. WebAssembly is abused by e-criminals. https://www.crowdstrike.com/blog/ecriminals-increasingly-use-webassembly-to-hide-malware/

[45] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1186–1189. https://doi.org/10.1109/ASE.2019.00133

[46] BoSun Park, JinGyo Song, and Seog Chung Seo. 2020. Efficient Implementation of a Crypto Library Using Web Assembly. *Electronics* 9, 11 (2020), 1839. https://www.mdpi.com/2079-9292/9/11/1839

[47] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. 2020. Verx: Safety verification of smart contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1661–1677. https://doi.org/10.1109/SP40000.2020.00024

[48] David A Ramos and Dawson Engler. 2015. Under-constrained symbolic execution: Correctness checking for real code. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*. 49–64.

[49] Sandra Rapps and Elaine J. Weyuker. 1985. Selecting Software Test Data Using Data Flow Information. *IEEE Trans. Softw. Eng.* 11, 4 (apr 1985), 367–375. https://doi.org/10.1109/TSE.1985.232226

[50] Nicola Ruaro, Kyle Zeng, Lukas Dresel, Mario Polino, Tiffany Bao, Andrea Continella, Stefano Zanero, Christopher Kruegel, and Giovanni Vigna. 2021. SyML: Guiding symbolic execution toward vulnerable states through pattern learning. In *24th International Symposium on Research in Attacks, Intrusions and Defenses*. 456–468. https://doi.org/10.1145/3471621.3471865

[51] Vaibhav Sharma, Soha Hussein, Michael W Whalen, Stephen McCamant, and Willem Visser. 2020. Java Ranger: Statically summarizing regions for efficient symbolic execution of Java. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 123–134. https://doi.org/10.1145/3368089.3409734

[52] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. 2016. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 138–157. https://doi.org/10.1109/SP.2016.17

[53] Amritraj Singh, Reza M Parizi, Qi Zhang, Kim-Kwang Raymond Choo, and Ali Dehghantanha. 2020. Blockchain smart contracts formalization: Approaches and challenges to address vulnerabilities. *Computers & Security* 88 (2020), 101654. https://doi.org/10.1016/j.cose.2019.101654

[54] Sunbeom So, Seongjoon Hong, and Hakjoo Oh. 2021. SMARTEST: Effectively Hunting Vulnerable Transaction Sequences in Smart Contracts through Language Model-Guided Symbolic Execution. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*.

[55] srdja. 2022. Collections-C, a library for data structures in C. https://github.com/srdja/Collections-C

[56] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting fuzzing through selective symbolic execution.. In *NDSS*, Vol. 16. 1–16. http://dx.doi.org/10.14722/ndss.2016.23368

[57] David Trabish, Andrea Mattavelli, Noam Rinetzky, and Cristian Cadar. 2018. Chopped symbolic execution. In *Proceedings of the 40th International Conference on Software Engineering*. 350–360. https://doi.org/10.1145/3180155.3180251

[58] Vladimír Vondruš. 2022. Graphics middleware libraries. https://github.com/mosra/magnum

[59] w3reality. 2021. Multithreading library for Rust and WebAssembly. https://github.com/w3reality/wasm-mt

[60] Dong Wang, Bo Jiang, and WK Chan. 2020. WANA: Symbolic Execution of Wasm Bytecode for Cross-Platform Smart Contract Vulnerability Detection. *arXiv preprint arXiv:2007.15510* (2020).

[61] Fish Wang and Yan Shoshitaishvili. 2017. Angr-the next generation of binary analysis. In *2017 IEEE Cybersecurity Development (SecDev)*. IEEE, 8–9. https://doi.org/10.1109/SecDev.2017.14

[62] WebAssembly. 2021. WebAssembly Official Site. https://webassembly.org/

[63] WebAssembly. 2022. A standard interface between WebAssembly and external environments. https://wasi.dev/

[64] Xusheng Xiao, Sihan Li, Tao Xie, and Nikolai Tillmann. 2013. Characteristic studies of loop problems for structural test generation via symbolic execution. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 246–256. https://doi.org/10.1109/ASE.2013.6693084

[65] Tao Xie, Nikolai Tillmann, Jonathan De Halleux, and Wolfram Schulte. 2009. Fitness-guided path exploration in dynamic symbolic execution. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*. IEEE, 359–368. https://doi.org/10.1109/DSN.2009.5270315

[66] Hui Xu, Zirui Zhao, Yangfan Zhou, and Michael R Lyu. 2018. Benchmarking the capability of symbolic execution tools with logic bombs. *IEEE Transactions on Dependable and Secure Computing* 17, 6 (2018), 1243–1256. https://doi.org/10.1109/TDSC.2018.2866469

[67] Guowei Yang, Corina S. Păsăreanu, and Sarfraz Khurshid. 2012. Memoized Symbolic Execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis* (Minneapolis, MN, USA) *(ISSTA 2012)*. Association for Computing Machinery, New York, NY, USA, 144–154. https://doi.org/10.1145/2338965.2336771

[68] Junfeng Yang, Can Sar, Paul Twohey, Cristian Cadar, and Dawson Engler. 2006. Automatically generating malicious disks using symbolic execution. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*. IEEE, 15–pp. https://doi.org/10.1109/SP.2006.7

[69] Yao Yao, Wei Zhou, Yan Jia, Lipeng Zhu, Peng Liu, and Yuqing Zhang. 2019. Identifying privilege separation vulnerabilities in IoT firmware with symbolic execution. In *European Symposium on Research in Computer Security*. Springer, 638–657. https://doi.org/10.1007/978-3-030-29959-0_31

[70] Qiuping Yi, Zijiang Yang, Shengjian Guo, Chao Wang, Jian Liu, and Chen Zhao. 2015. Postconditioned symbolic execution. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 1–10. https://doi.org/10.1109/ICST.2015.7102601

[71] Qiuping Yi, Zijiang Yang, Shengjian Guo, Chao Wang, Jian Liu, and Chen Zhao. 2017. Eliminating path redundancy via postconditioned symbolic execution. *IEEE Transactions on Software Engineering* 44, 1 (2017), 25–43. https://doi.org/10.1109/TSE.2017.2659751

Received 2023-02-16; accepted 2023-05-03