Automata-Guided Control-Flow-Sensitive Fuzz Driver Generation

Cen Zhang and Yuekang Li, *Nanyang Technological University, Continental-NTU Corporate Lab;* Hao Zhou, *The Hong Kong Polytechnic University;* Xiaohan Zhang, *Xidian University;* Yaowen Zheng, *Nanyang Technological University, Continental-NTU Corporate Lab;* Xian Zhan, *Southern University of Science and Technology; The Hong Kong Polytechnic University;* Xiaofei Xie, *Singapore Management University;* Xiapu Luo, *The Hong Kong Polytechnic University;* Xinghua Li, *Xidian University;* Yang Liu, *Nanyang Technological University, Continental-NTU Corporate Lab;* Sheikh Mahbub Habib, *Continental AG, Germany*

https://www.usenix.org/conference/usenixsecurity23/presentation/zhang-cen

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Automata-Guided Control-Flow-Sensitive Fuzz Driver Generation

Cen Zhang[†] Yuekang Li [†] [*] Hao Zhou [‡] Xiaohan Zhang [¶] Yaowen Zheng [†] Xian Zhan [§‡]
Xiaofei Xie [£] Xiapu Luo [‡] Xinghua Li [¶] Yang Liu [†] Sheikh Mahbub Habib [††]

[†]Nanyang Technological University, Continental-NTU Corporate Lab [‡] The Hong Kong Polytechnic University [¶] Xidian University
[£] Singapore Management University [§] Southern University of Science and Technology [††] Continental AG, Germany

Abstract

Fuzz drivers are essential for fuzzing library APIs. However, manually composing fuzz drivers is difficult and time-consuming. Therefore, several works have been proposed to generate fuzz drivers automatically. Although these works can learn correct API usage from the consumer programs of the target library, three challenges still hinder the quality of the generated fuzz drivers: 1) How to learn and utilize the control dependencies in API usage; 2) How to handle the noises of the learned API usage, especially for complex real-world consumer programs; 3) How to organize independent sets of API usage inside the fuzz driver to better coordinate with fuzzers.

To solve these challenges, we propose RUBICK, an automata-guided control-flow-sensitive fuzz driver generation technique. RUBICK has three key features: 1) it models the API usage (including API data and control dependencies) as a deterministic finite automaton; 2) it leverages active automata learning algorithm to distill the learned API usage; 3) it synthesizes a single automata-guided fuzz driver, which provides scheduling interface for the fuzzer to test independent sets of API usage during fuzzing. During the experiments, the fuzz drivers generated by RUBICK showed a significant performance advantage over the baselines by covering an average of 50.42% more edges than fuzz drivers generated by FUZZGEN and 44.58% more edges than manually written fuzz drivers from OSS-Fuzz or human experts. By learning from large-scale open source projects, RUBICK has generated fuzz drivers for 11 popular Java projects and two of them have been merged into OSS-Fuzz. So far, 199 bugs, including four CVEs, are found using these fuzz drivers, which can affect popular PC and Android software with dozens of millions of downloads.

1 Introduction

Fuzzing is a practical dynamic analysis technique for vulnerability detection. Compared with static analysis techniques, fuzzing can produce more precise results with few false-positives and provide security analysts with Proof-of-Concept inputs to replay the bugs. However, as a dynamic technique, fuzzing requires an executable of the target software as the testing subject. To fuzz libraries, executable programs using the library functions must be generated. Conventionally, these programs are called fuzz harnesses or fuzz drivers.

Fuzz drivers can be composed manually by human experts or generated automatically by tools. To compose fuzz drivers, experts have to learn the usage of library APIs from documentations or example programs. Not only is this learning process tedious and time-consuming, but also the quality of the composed fuzz drivers heavily depends on the experience of the human experts. Therefore, techniques for automatically generating fuzz drivers are needed.

Several existing works [1–3] focus on automatic fuzz driver generation. Similar to manually writing the fuzz drivers, these techniques also need to learn the correct usage of library APIs to ensure testing effectiveness. Additionally, most existing works learn such knowledge by analyzing how the consumer programs (aka example programs) of the libraries utilize the API functions. After generating the fuzz drivers, these techniques also need to rank [2] or ensemble [1] the generated fuzz drivers so that the fuzzer can test them substantially.

Despite the previous efforts, three challenges still exist for both learning the correct usage of library APIs and utilizing the generated fuzz drivers: **C1.** Some library APIs should reside in branches and loops where the conditions are guarded by the results of other APIs, but the API usage learned by existing works emphasizes on data dependencies among the APIs while ignoring most control dependencies among them. Not including the control dependencies can end up in failing to invoke certain APIs properly. **C2.** The learned API usage of existing works suffers from noises such as redundant API usages or wrong API dependencies. Failing to remove the noises can limit the usability of a fuzz driver generation technique especially when it needs to learn from complex real-world consumer programs. **C3.** Multiple fuzz drivers can be generated for a single target library with existing works, but how

[*]Corresponding Author.

to organize and utilize these fuzz drivers to guarantee that they can be substantially tested by the fuzzer is understudied. Poorly organized fuzz drivers can distract the fuzzer and hinder the fuzzing performance.

To address the three challenges, we propose RUBICK [1] — an automata-guided control-flow-sensitive fuzz driver generation technique. **C1.** The rationale of RUBICK is that API control dependencies such as branches and loops can be represented as automatons. By properly defining the events (i.e., the alphabet of an automaton), API control dependencies are interpreted as event sequences (i.e., accepted strings of an automaton). Based on this modeling, extraction algorithms are designed to extract usage automatons from API consumers. **C2.** Since an automaton intrinsically represents a set of accepting event sequences, denoising extracted usages means generating a minimized automaton which only accepts valid sequences. RUBICK adapts L*, an active automata learning algorithm, to accomplish this goal. By defining what is a valid sequence (membership queries, abbr as MQ) and what is an acceptable automaton (equivalence query, abbr as EQ), the algorithm starts from an empty automaton and iteratively improves that automaton using feedbacks from MQ and EQ until EQ is satisfied. RUBICK combines static and dynamic information to answer the validity of sequences. And the automaton is acceptable when it does not falsely accept or reject sequences. Each extracted automaton is denoised separately and later merged together with others as one usage automaton. **C3.** Note that the usage automaton may contain multiple independent sets of API usages. Instead of generating multiple fuzz drivers, RUBICK generates a single automata-guided fuzz driver. It provides a scheduling interface that fuzzers can pick the testing usage set by mutating specific bytes of input. By doing so, the utilization of independent API usage sets are benefited by existing seed schedulers inside fuzzers.

In evaluation, the fuzz drivers of RUBICK are compared with drivers from FUZZGEN [2] and from OSS-Fuzz [4] or other human experts on six popular Java projects. The results show that RUBICK outperforms its competitors in both code coverage (on avg. 50.42%, 44.58% more edge coverage than FUZZGEN, manually written fuzz drivers) and unique bugs (on avg. 45.92%, 98.59% more unique bugs than FUZZGEN, manually written fuzz drivers). RUBICK has generated fuzz drivers for 11 open source projects and two of them have been merged into OSS-Fuzz. So far, 199 bugs (four CVEs) have been found using these fuzz drivers. These bugs are of popular Java projects such as Apache Software Foundation, which affects the PC and Android software with dozens of millions of downloads.

In summary, our contributions are:

- We identified three key challenges for generating fuzz drivers from consumer programs. Besides, we proposed an automata-based solution to solve these challenges;
- We implemented RUBICK as the first tool which can learn API usage from large-scale open source projects and generate control-flow-sensitive fuzz drivers;
- We applied RUBICK to 11 popular Java projects and discovered 199 bugs (four CVEs). We responsibly disclosed them and helped the vendors to fix them.

RUBICK is open-source for facilitating future research [5].

[1] RUBICK is a Dota 2 hero who can learn spells from enemies and cast them more powerfully.

[2] FUZZGEN is the most related work and we implemented its Java version.

2 Preliminaries

2.1 Backgrounds

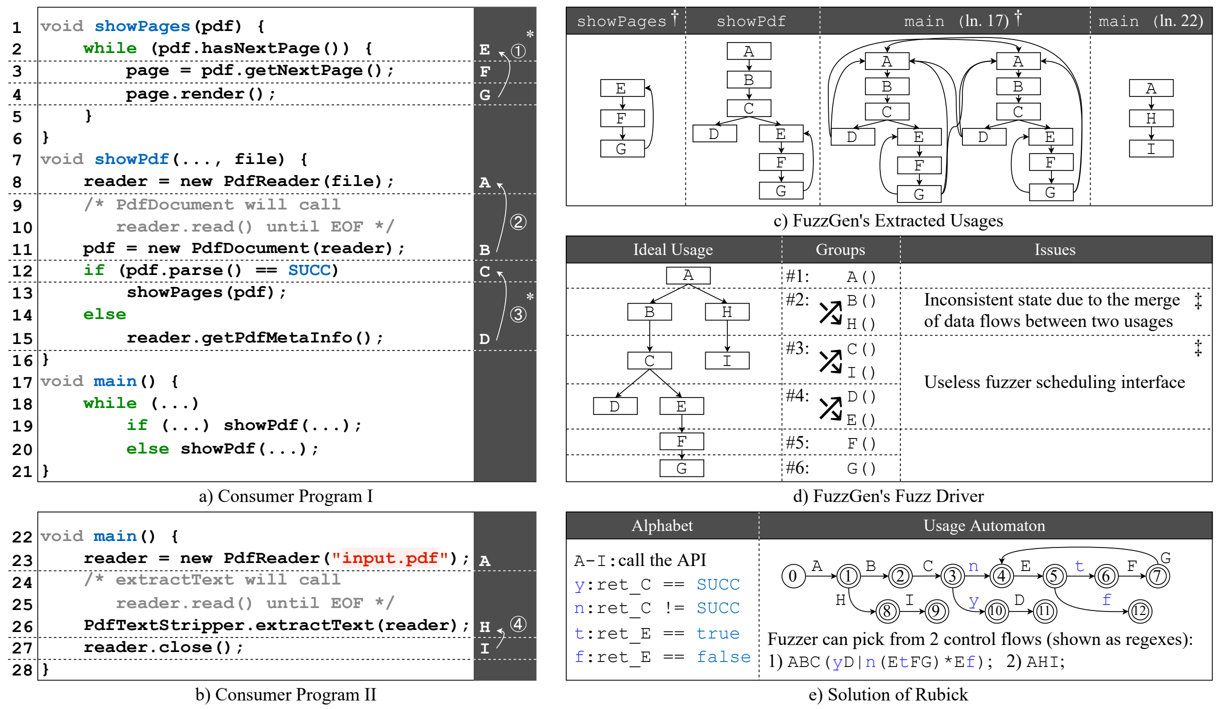
Deterministic Finite Automaton We use Deterministic Finite Automaton (DFA) to model the API usage. DFA contains five elements: a finite set of states Q, a set of input symbols (aka letters) called the alphabet Σ, a transition function δ: $Q \times \Sigma \rightarrow Q$, an initial state $q_0 \in Q$, and a set of final states $F \subseteq Q$. Intrinsically, a DFA is an acceptor of strings, i.e., sequences of letters. Any sequence corresponding to a path from the initial state to a final state is accepted by that DFA.

Automata Learning Algorithm Generally, the algorithm learns an automaton from a set of positive examples (represents for valid strings, abbr as PE) and negative examples (represents for invalid strings, abbr as NE). There are two types of the algorithms: passive learning and active learning. The former requires a finite set of PEs and NEs of the system under learning (SUL) before learning, whereas the latter finds PEs and NEs by asking teacher questions about the SUL during the learning. Passive learning builds output automaton based on the given learning input. It usually has adequate time complexity but its performance heavily depends on the representativeness of the learning input. For active learning, it requires a teacher to answer two kinds of queries of the SUL: the membership query (MQ) and the equivalence query (EQ). The membership query asks about the validity of a string, i.e., a given string is of PE or NE. And the equivalence query asks whether the learned automaton is equivalent to the final answer. If the answer is no, the teacher also needs to return a counterexample (a falsely accepted/rejected string) as feedback. The whole process of active learning is that: the algorithm starts from an empty automaton, then it iteratively improves that automaton using feedback from MQ and EQ until the EQ is satisfied. Active learning algorithms, e.g., L* [6], have a good learning performance (the learned automaton is minimized and accurate) but suffer from its exponential learning costs.

2.2 Challenges of Existing Works

Fig. 1 illustrates the three challenges of building a desired fuzz driver using an example simplified from real-world cases.

C1: Modeling of API Control Dependencies The first challenge is how to model and learn the control dependencies among the library APIs. APIs can have different types



```
1  void showPages(pdf) {
2      while (pdf.hasNextPage()) {
3          page = pdf.getNextPage();
4          page.render();
5      }
6  }
7  void showPdf(..., file) {
8      reader = new PdfReader(file);
9      /* PdfDocument will call
10        reader.read() until EOF */
11     pdf = new PdfDocument(reader);
12     if (pdf.parse() == SUCC)
13         showPages(pdf);
14     else
15         reader.getPdfMetaInfo();
16 }
17 void main() {
18     while (...)
19         if (...) showPdf(...);
20         else showPdf(...);
21 }
```

a) Consumer Program I

```
22 void main() {
23     reader = new PdfReader("input.pdf");
24     /* extractText will call
25        reader.read() until EOF */
26     PdfTextStripper.extractText(reader);
27     reader.close();
28 }
```

b) Consumer Program II

c) FuzzGen's Extracted Usages

Ideal Usage	Groups	Issues
	#1: A()	
	#2: B() H()	Inconsistent state due to the merge of data flows between two usages ‡
	#3: C() I()	Useless fuzzer scheduling interface ‡
	#4: D() E()	
	#5: F()	
	#6: G()	

d) FuzzGen's Fuzz Driver

Alphabet	Usage Automaton
A-I:call the API y:ret_C == SUCC n:ret_C != SUCC t:ret_E == true f:ret_E == false	Fuzzer can pick from 2 control flows (shown as regexes): 1) ABC(yD|n(EtFG)*Ef); 2) AHI;

e) Solution of Rubick

Figure 1: Motivation Example. The letters A - I are used to represent the call of the API function which appears at the same line with the letter in subfigure a), b). The symbols *, †, and ‡ represent the example cases that existing works haven't solved in C1, C2, C3 respectively. In subfigure d), #1 - #6 represent group 1 - 6, and ⤨ represents the execution order of a group's functions depends on the input (i.e., fuzzer scheduling interface).

of dependencies. For example, in Fig. 1.a) and Fig. 1.b), ② and ④ are the explicit data dependency between A and B and the implicit data dependency between H and I, respectively. ① and ③ are two types of control dependencies (representing loop and branch). Unfortunately, limited by how they model the API usage, existing works do not consider most control dependencies. FUZZGEN [1] uses flattened A^2DG (groups of API call sequences) to describe the usage and APICRAFT [2] uses a data dependency tree. Both models are not aware of the control dependencies ① and ③. As a compensation, they proposed heuristics to locate the error branches (by recognizing the call of signature functions such as `exit`), which covers a special case of the control dependency. Though not emphasized in previous works, control dependencies can significantly affect the quality of the generated fuzz drivers. For example, in Fig. 1.a), missing the control dependencies can cause not only the improper API invocation (`showPages` can get invoked even when the program failed to parse the input if missing ③), but also the insufficient exploitation of the input (the program will not parse the second page onwards if missing ①). Consequently, to learn and utilize control dependencies, a more descriptive model is needed.

C2: Noises in Learned Usage The second challenge is the learned API usage can be full of noises while learning from real world consumer programs (e.g., open-source projects in GitHub). The noises are introduced from the following scenarios: ❶ The learned usage can be incomplete or redundant when the learning starts from imperfect entry points of the program. For instance, Fig. 1.c) shows the cases of applying FUZZGEN to the consumer programs in Fig. 1.a) and Fig. 1.b). The extracted usage of `showPages` and `main` (line 17) is incorrect. For `showPages`, the usage is incomplete since the source of the `pdf` is missing. For `main` (line 17), the usage has unnecessary complexity. In practice, starting the extraction from `main` (the default strategy of FUZZGEN) usually incurs inefficient or even incorrect results since the extracted usages can be too complex to be used; ❷ Extracted usage can be erroneous due to the coupling of the usage code and the consumer program code. For example, an usage irrelevant loop wrapping around an API in the consumer program may add a dead loop to the extracted usage. ❸ The usage can be incorrect due to the imprecise analysis. Failing to remove the noises not only wastes fuzzing resources but also introduces false positives during fuzzing. Unfortunately, existing works do not handle the noises. Considering the ubiquity of the noises and the vague boundaries between the noise and the usage, denoise techniques should be introduced.

C3: Utilization of Independent Usage Scenarios The third challenge is how to effectively organize independent API usage scenarios during fuzzing so that the fuzzer can substantially test the APIs. Here we call a set of self-sustaining API functions an *usage scenario*. For example, in Fig. 1.c), the APIs extracted from `showPdf` forms one usage scenario while the APIs extracted from `main (ln.22)` forms another.

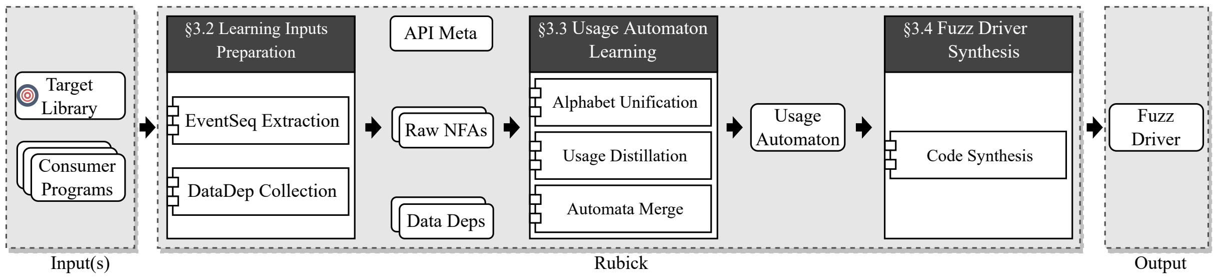


Figure 2: General Workflow of RUBICK.

Usually, more than one usage scenario can be learned from the consumer programs and each one can be converted as one standalone fuzz driver. The reason why there exist multiple independent usage scenarios is two-fold. On the one hand, for better usability or compatibility, developers may provide multiple sets of APIs for one functionality to users. On the other hand, the learned usages are influenced by consumer program specifics, which cannot be thoroughly merged as one usage scenario without additional domain knowledge. In fact, even human experts who wrote fuzz drivers for OSS-Fuzz get confused about what is the proper way to handle this problem [7]. Besides, existing works have not thoroughly tackled this problem. FUZZGEN partially solves this problem by proposing algorithms to coalescence distinct usage scenarios (merging the common nodes between two scenarios) and providing scheduling interface to fuzzer. However, as shown in Fig. 1.d), the coalesce of two distinct control flows is error-prone due to the inconsistency of their data flows (Both `B` and `H` depend/modify reader's status.) Besides, scheduling function orders inside one group (FUZZGEN divides them into groups using relaxed top sort) tends to be unnecessary (Reordering `D` and `E` does not help fuzzing, so does `C` and `I`.) Other works, such as [2], rely on users to select the fuzz drivers. The fuzzing performance can be hindered without proper utilization strategy since some usage scenarios may get starved during fuzzing. Therefore, a better utilization strategy of usage scenarios should be proposed.

2.3 Our Approach

The key observation is that, by properly defining events, common API dependencies affecting the control flow can be modeled as an automaton accepting certain event sequences. For example, to describe API usages of Fig. 1.a) and Fig. 1.b), we define *function events* (marked by `A - I`) and *condition events* (marked by `y`, `n`, `t`, `f`). The full definition is shown in the Alphabet part of Fig. 1.e), and the right side of Fig. 1.a) and Fig. 1.b). A function event represents the call of an API function, e.g., event `C` means the call of `PdfDocument.parse`. A condition event means that a constraint regarding the return values of API functions has been satisfied, e.g., event `y` represents the last return value of `PdfDocument.parse` equals to `SUCC`. For simplicity, the automata representing the usage are written in Python regex syntax [8]. Using the above definition, the loop in Fig. 1.a) line 2 - 5 can be written as `(EtFG)*Ef`, and the branch in line 12 - 15 can be represented as `yD|n(EtFG)*Ef`. Similarly, the API usages of Consumer Program I and II are `ABC(yD|n(EtFG)*Ef)` and `AHI`.

Based on this observation, RUBICK proposes an automata-based solution to solve the previously discussed challenges. Firstly, RUBICK extracts automatons from consumer programs. The extracted automatons contain control-flow-sensitive API usage but are raw. Secondly, it adapts an active learning algorithm to find a minimized automaton which removes both the duplicate and invalid event sequences inside each automaton. After denoising, RUBICK merges them together as one single usage automaton. Lastly, to better utilize this usage automaton, it synthesizes an automata-guided fuzz driver. The driver provides a scheduling interface which fuzzers can pick the testing usage scenario by mutating specific bytes of input. Therefore, the usage scenarios inside the fuzz driver can be scheduled by existing seed schedulers inside fuzzers. Fig. 1.e) shows the usage automaton RUBICK learned from Consumer Program I and II. Fuzzers can choose to fuzz either usage scenario.

Accordingly, as shown in Fig. 2, RUBICK has three components: ❶ Learning Materials Preparation. The materials include the event sequences in Raw non-deterministic finite automaton (NFA) format, the API data dependencies, and the API meta information; ❷ Usage Automaton Learning. RUBICK first unifies the alphabet for all Raw NFAs. Then it uses a L* [6] based algorithm to generate a distilled (aka denoised) DFA for each Raw NFA. After distillation, RUBICK merges these DFAs as one usage automaton; ❸ Fuzz Driver Synthesis. RUBICK synthesizes an automata-guided fuzz driver.

3 Methodology

3.1 API Usage Modeling

API Usage Representation RUBICK uses usage automaton to represent API usage. Usage automaton contains two kinds of events: the function events and the condition events. In the context of the automaton, a function can bind with zero or more output variables. And a condition expression

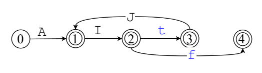
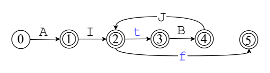
	Loop Case I	Loop Case II
Code	`it = pdf.pageIterator();` `while (it.hasNext())` `it.getNext();`	`n = pdf.getPageNum();` `for (i = 0; i < n; i = i + 1)` `pdf.getPage(i);`
Mock APIs	`bool mock_I(Iterator it)` `{ return it.hasNext(); }` `PdfPage mock_J(Iterator it)` `{ return it.getNext(); }`	`int mock_I()` `{ return 0; }` `int mock_J(int i)` `{ return i + 1; }`
Alphabet	`A:ret_A = pdf.pageIterator();` `I:ret_I = mock_I(ret_A);` `J:ret_J = mock_J(ret_A);` `t:ret_I == true;` `f:ret_I == false;`	`A:ret_A = pdf.getPageNum();` `B:ret_B = pdf.getPage(ret_I);` `I:ret_I = mock_I();` `J:ret_I = mock_J(ret_I);` `t:ret_I < ret_A;` `f:ret_I >= ret_A;`
Automaton		

Figure 3: Loop Cases Requiring Extra Events. The usage of I and II are `A(ItJ)*If` and `AI(tBJ)*f` respectively (represented in regexes).

consists of constants and output variables. A function event represents an action calling an API function and updating the value of its bound output variables. A condition event represents an action that a condition expression is evaluated as true. Naively, using API functions to define function events and condition events, all sequential and branch usage can be described. However, the above events can only describe the loop whose control variables' data flow (initialization, increment, and condition) can be fully recorded by these events. For example, the loop in Fig. 1.a) is fully recorded using `E`, `G`, `t`, `f`. Fig. 3 shows two types of partially recorded loops. For case I, the condition and increment of `it` are missed since `it.hasNext()` and `it.getNext()` are not API functions. For case II, the initialization and increment of loop's control variable `i` are missed since `i = 0`, `i = i + 1` are not recorded. Mock APIs are introduced to describe the missing data flow of control variables. Section 3.2.1 discusses how to locate them. Specifically, in case I, by introducing mock APIs `mock_I` and `mock_J`, new function events `I`, `J` and condition events `t`, `f` (tainted by `ret_I`) are identified. Therefore, the loop usage in case I can be fully described as `A(ItJ)*If`. Similarly, the usage in case II can be written as `AI(tBJ)*f`.

Properties of Usage Automaton **P1**: No initial state can be a final state. This is because RUBICK treats empty usage as invalid usage. **P2**: Any non-empty prefix of a valid event sequence is a valid event sequence. For example, in Fig. 1.b), given that `AHI` is valid, obviously `AH` is valid too. **P3**: Any event sequence containing a non-empty invalid prefix is invalid. For instance, since `AIH` is invalid, any event sequence starting with `AIH` is also invalid. This means that all invalid strings lead the state machine into a trap state [9]. **P2** and **P3** are useful in inferring the validity of event sequences which can boost the distillation process in Section 3.3.2.

3.2 Learning Inputs Preparation

RUBICK collects learning inputs from consumer programs via static analysis. Note that RUBICK requires no a priori

Algorithm 1 Raw Usage Automatons Extraction

```
Input: $C$ (Consumer Program)
Output: $As$ (Raw Usage Automatons)
 1: procedure EXTRACT-USAGE($iCFG, F, ctxt$)
 2:     $A, Q, insn2State \leftarrow$ empty nfa, [], { }
 3:     $startS, endS \leftarrow new\ State(), new\ AcceptState()$
 4:     for $Insn_{entry} \in Get\text{-}Entries(F)$
 5:         $Q \overset{+}{\leftarrow} [\ \langle\ startS, Insn_{entry}, ctxt.clone()\ \rangle\ ]$
 6:     while $Q$ is not empty
 7:         $curS, curI, ctxt \leftarrow Q.pop()$
 8:         if $curI$ is a non-API func call instruction
                ▷ Extract and merge $subA$
 9:             $F_{callee} \leftarrow iCFG.getCallee(curI)$
10:             $subA \leftarrow EXTRACT\text{-}USAGE(ICFG, F_{callee}, ctxt)$
11:             $curS, A \leftarrow Merge\text{-}SubNFA(curS, A, subA)$
12:         else
                ▷ Add transitions to identified new events to $A$
13:             $ctxt, event \leftarrow Event\text{-}Identification(ctxt, curI)$
14:             if $event \neq null$
15:                 $nextS \leftarrow new\ AcceptState()$
16:                 $A \overset{+}{\leftarrow} new\ Transition(curS, nextS, event)$
17:                 $curS \leftarrow nextS$
18:         $insn2State \overset{+}{\leftarrow} \{\ curI : curS\ \}$
19:         if $curI$ has no succs
20:             $A \overset{+}{\leftarrow} new\ Transition(curS, endS, \epsilon)$
21:         else
22:             for $nextI \in iCFG.getSuccs(curI)$
23:                 if $nextI \in insn2State$
24:                     $nextS \leftarrow insn2State.get(nextI)$
25:                     $A \overset{+}{\leftarrow} new\ Transition(curS, nextS, \epsilon)$
26:                 else
27:                     $Q \overset{+}{\leftarrow} \langle\ curS, nextI, ctxt.clone()\ \rangle$
28:     return $A$
29: end procedure
30: $As, ICFG \leftarrow [], Get\text{-}ICFG(C)$
31: for $F_{target} \in Get\text{-}Functions(C)$
32:     $As \overset{+}{\leftarrow} EXTRACT\text{-}USAGE(ICFG, F_{target}, empty\ context)$
```

knowledge about the target library. Given a library, RUBICK collects: ❶ the candidate events and event sequences; ❷ the API data dependencies; ❸ the API meta information. The API meta information refers to the basic information of API functions, e.g., the function signature, the type of arguments and return value, etc. Its collection is straightforward and is done once per library. In the following, we only detail the collection of the learning inputs ❶ and ❷.

3.2.1 Event Sequence Extraction

RUBICK extracts event sequences from the consumer programs by converting its control flow graph (CFG) into a non-deterministic finite automata (NFA). Generally this is done by translating some instructions into events and removing the irrelevant instructions. We first discuss how events are identified from instructions, then explain the extraction algorithm.

Events Identification The identification of API function related events is straightforward: an instruction calling any API function is a function event, and any branch instruction whose condition expression is tainted by output of API functions is a pair of condition events (true and false branches). For the identification of mock API related events, RUBICK needs to first locate the mock APIs and then apply the above identification. RUBICK identifies the mock APIs of the two types of loop cases discussed in Section 3.1 separately. For case I, RUBICK additionally models the iterator interface. The instructions which call iterator functions with a tainted class object will be identified as mock API functions. For case II, the identification is based on the results of the extraction. Hence, RUBICK may run the extraction twice. After first run, RUBICK checks whether the expressions of any extracted loop condition contain non-output variables. If no one contains such variables, the extraction process is complete and all loops are fully represented by the events. Otherwise, the data flows of control variables for loop conditions with non-output variables are not fully recorded. Therefore, RUBICK conducts data dependency analysis to locate the instructions which initialize or update that variable. After setting these instructions as mock API functions, RUBICK reruns the extraction for final results.

Raw NFA Extraction For clarity, Algorithm 1 shows a single-pass extraction process. As shown in line 31 – 32, instead of requiring a perfect entry function for extraction, RUBICK extracts a raw usage automaton (namely Raw NFA) for each function inside the consumer program. This is feasible since the subsequent learning process will remove the invalid or the redundant usages. The basic idea is to build the Raw NFA along the traverse of the ICFG. The traverse starts from the entry instruction of target function F_{target}. Line 7 – 27 shows the analysis of each traversed instruction. $curI$ points to the instruction under analysis. $curS$ points to the state in Raw NFA which new states should be linked with. $ctxt$ holds the taint information for analyzing $curI$. If $curI$ is a non-API function call instruction, RUBICK extracts the Raw NFA of the callee ($subA$) and merges it into current NFA (A). The merge is accomplished by adding transition from $curS$ to the start state ($startS$) of $subA$ with epsilon event (ϵ) and adjusting the $curS$ to point to the end state ($endS$) of $subA$. If $curI$ is not of the above case, the event identification strategies are applied. New transition will be added to A once a new event is identified, and $curS$ also will be updated. Last, the successors of $curI$ are added to Q for analysis. If $curI$ is an exit point of the function , e.g., `return`, the edge from $curS$ to $endS$ is added. If a successor instruction is already analyzed, a transition from $curS$ to the instruction's corresponding state under event ϵ is added. For simplicity, the algorithm only shows the key flow. In implementation, the algorithm also maintains a stack to prevent the infinite loop caused by the recursive call of the target function. Besides, multiple callees can be returned by *iCFG.getCallee* (ICFG stands for Interprocedural Control Flow Graph) in line 9 when $curI$ is an indirect call. RUBICK empirically picks the first callee. If the callee is picked wrongly, the extracted wrong usage is expected to be filtered by usage distillation. The extracted automaton is named as Raw NFA since it contains ϵ and can have multiple transitions given one specific state and event, e.g., same events can be identified in both paths of a branch.

3.2.2 Data Dependency Collection

The extracted Raw NFA contains control flow information of the fuzz driver, such as the correct order of API functions, or the condition to call an API function, etc. Comparatively, API data dependencies indicate the possible values for arguments of API functions, or the data linkages between API functions. Both are necessary information for generating a valid fuzz driver. Specifically, one set of control flow usage can have multiple sets of data dependencies. For example, given three API functions F_A, F_B, F_C, and assuming the return values of both F_A and F_B can be used as the first argument of F_C, the control flow usage $F_A \rightarrow F_B \rightarrow F_C$ has two sets of data dependencies: F_C can use the return value of either F_A or F_B. Currently, RUBICK collects data dependencies between two API functions and between an API function and a constant. Specifically, RUBICK abstracts them as the tuple $\langle$ Provider, Consumer $\rangle$ where provider can be any output of an API function or a constant and the consumer can be any input needed by an API function. They are collected together with Raw NFAs by statically analyzing the consumer programs.

3.3 Usage Automaton Learning

3.3.1 Alphabet Unification

The collected Raw NFAs have their own alphabets. RUBICK unifies them as one alphabet by identifying the equivalent letters and assigning them the same letter. ❶ For API function events, RUBICK assigns same letter for events have same function signatures. ❷ For condition events, RUBICK needs to align their condition expressions and solve the potential conflicts before assigning letters. For example, assuming Raw NFA A has C_{A1}: `ret_C == SUCC`, C_{A2}: `ret_C != SUCC`, and Raw NFA B contains C_{B1}: `ret_C == SUCC`, C_{B2}: `ret_C == STOP`. Ideally RUBICK can use the following letters to replace all above letters: C_{U1}: `ret_C == SUCC`, C_{U2}: `ret_C == STOP`, C_{U3}: `(ret_C != SUCC) && (ret_C != STOP)`. For instance, all edges representing event C_{A2} inside NFA A can be replaced with two edges representing C_{U2} and C_{U3}. RUBICK models this conflict solving problem as a solution set division problem. Generally, each condition expression is equivalent to its solution set satisfying the condition. The goal is to find a set of solution sets where each solution set is non-intersect with each other and any original solution set can be the union of them. Using that set, any original condition event can be replaced by several new events (see Appendix A for the algorithm detail). From our experience, conflict case rarely happens and RUBICK only met conflict of simple conditions containing single variable expressions. ❸ For the mock API function related events, RUBICK compares their identity by group. Specifically, a group of events contain all related events affecting the loop condition. Two groups are identical if their loop condition expressions, the value update expression of the mocked API functions, and the execution order of these events are equal. Otherwise, RUBICK assigns different sets of letters to them.

3.3.2 Usage Distillation

The extracted Raw NFAs can contain invalid event sequences. The goal of usage distillation is to find a minimized automaton which only accepts the correct event sequences inside a NFA. Automata learning algorithms fit this task. Note that the Raw NFA only provides the scope of PEs but not the exact sets. Besides, the NFA can contain infinite amounts of PEs and NEs. In this case, active learning is more suitable since sampling a representative set of learning input for passive learning is challengeable. If we gather the knowledge related with validating event sequences to build a teacher, it can learn a minimized DFA to represent the SUL (the correct parts of the Raw NFA). The knowledge includes the Raw NFA, the static checker and the dynamic validator for event sequences. For the active learning algorithm, RUBICK uses L*[6].

To build a teacher required by L*, RUBICK has to answer two types of queries: membership query and equivalence query. For membership query, RUBICK needs to answer the validity of any queried event sequence. RUBICK validates a sequence both statically and dynamically (see Algorithm 2). ❶ In line 8, RUBICK checks whether the used variables of any event have been initialized in its prefix events. If the check fails, the sequence is invalid since the fuzz driver based on it will have uninitialized variables. ❷ In line 9, RUBICK converts the event sequence to a fuzz driver and executes it. The sequence is invalid if the execution fails (crash or stuck). Note that this fuzz driver is only a sequence of API calls and condition checks without branches or loops. Sometimes, the sequence can only be partially executed since the execution may not satisfy a condition when the sequence contains condition events. In this case, the validity is roughly measured by the executed parts. This is an optimistic strategy and can cause the final automaton containing invalid sequences. Obviously, the effectiveness of dynamic validation is influenced by the diversity of the input files. Practically we suggest using one to three valid inputs as seeds. In evaluation, RUBICK uses one valid seed (< 100K) downloaded from the Internet for each target. ❸ In line 7, RUBICK checks two properties: First, the condition event can only appear after its expression's value has been updated by other events. This helps to filter out the dead loop caused by condition events whose value will never be updated. Second, the API function accepting the input file should appear once

Algorithm 2 Membership Query Pseudo Code

```
Input: eventseq (A string represents an event sequence), rawNFA
    (an extracted Raw NFA)
Output: boolean (boolean value for eventseq is accepted or not)
 1: procedure MEMBERSHIP-QUERY(eventseq, rawNFA)
 2:     if $eventseq \in neCache$
 3:         return false
 4:     if $eventseq \in peCache$
 5:         return true
 6:     if $eventseq \in rawNFA$
 7:         if Fit-Properties-Of-Desired-Fuzz-Drivers(eventseq)
 8:             if No-Unsatisfied-Data-Dependency(eventseq)
 9:                 if Pass-Dynamic-Validation(eventseq)
10:                     $peCache \overset{+}{\leftarrow} eventseq$
11:                     return true
12:     $neCache \overset{+}{\leftarrow} eventseq$
13:     return false
14: end procedure
```

and only once. For the equivalence query, RUBICK uses wp-method [10] to sample a test set of the automaton. RUBICK selects wp-method since it generates a slightly smaller test set than w-method while keeping similar representativeness. The lookahead value is the only adjustable parameter in wp-method. During the EQ, the L* algorithm tries to search for test cases which are counterexamples. The test cases are generated by adding postfixes to the accepted strings. The lookahead value determines the maximum length of the added postfixes. Therefore, a higher lookahead value means a more complete check in EQ but brings more performance penalty. Empirically, RUBICK sets it as 2 (see evaluation in Section 4.2).

After learning, RUBICK removes dead loops and trap states of the output automaton. A dead loop is a loop which does not contain any condition event. Removing the trap states and the related transitions simplifies the automaton while keeping its accepted event sequences (**P3** in Section 3.1).

3.3.3 Automata Merge

RUBICK uses DFA combination and minimization algorithms to generate the final usage automaton from the distilled ones. These algorithms, such as Hopcroft's DFA minimization algorithm [11], have reasonable time complexity. The merge process cannot be done before distillation. The reason is that the merge will exponentially increase the performance costs (the distillation will face a giant Raw NFA) while gaining little benefits on distillation outcomes.

3.4 Automata-Guided Fuzz Driver Synthesis

As the last step, RUBICK synthesizes a fuzz driver based on the learned usage automaton. However, the synthesis is non-trivial since the learned usage can inevitably contain multiple independent usage scenarios (Section 2.2 C3). For better

Table 1: The Attack Surfaces Used for Evaluation. A: Apache Software Foundation, C: Commercial Company, G: Github Individual.

	Format	Project	Version	Vendor	# of APIs
apachetar	TAR	Apache Commons Compress [12]	1.21	A	195
apachepoi	XLS	Apache POI [13]	5.2.1	A	1,320
itextpdf	PDF	iText 7 [14]	7.2.2	C	810
junrar	RAR	Junrar [15]	7.4.1	G	619
pdfbox	PDF	Apache PDFBox [16]	2.0.26	A	2,779
zip4j	ZIP	Zip4j [17]	2.9.1	G	521

scheduling of fuzzing multiple scenarios, RUBICK synthesizes an automata-guided fuzz driver. It provides a scheduling interface where the fuzzer can pick the testing scenario by mutating the specific bytes of the input. Therefore, the scenario scheduling can be done by the existing seed schedulers inside fuzzers. For example, in Fig. 1.e), fuzzers can pick to fuzz 1) or 2) by mutating the first byte of the input. Fig. 9 in Appendix C details the implementation of our fuzz driver.

Generally, RUBICK first uses a Depth-First-Search (DFS) based algorithm to count all independent usage scenarios inside the automaton (see detail in Appendix B). Second, the fuzz driver is designed as event-driven. It loads the usage automaton and maintains its states during execution. For each execution, it starts from the initial state and tries to traverse the automaton until there are no successor state. After each step of the traverse, the driver executes the corresponding code, and updates the status accordingly. During the traverse, when there are multiple choices for picking the next states, the pick is determined by either the mutated input or the execution context. If the next events belong to multiple usage scenarios, the pick is determined by input, e.g., for state 1 in Fig. 1.e), choosing `B` or `H` depends on the input. Otherwise, it should be determined by the execution context, e.g., for state 3 in Fig. 1.e), choosing `y` or `n` depends on the value of `ret_E`.

4 Implementation & Evaluation

Implementation The main components of RUBICK contain 6,979 lines of Java code, 1,656 lines of Python code, and 654 lines of Bash scripts. Specifically, Java code includes most functionalities such as the learning inputs collection, usage distillation, fuzz driver synthesis, etc. The python code is mainly used for alphabet unification. The bash scripts are used for gluing the workflow. The automata related algorithms are developed upon `learnlib` [18], and the first-order logical formulas related algorithms are developed upon `z3py` [19]. Currently, RUBICK can generate fuzz drivers for Java libraries.

Evaluation Questions The evaluation aims to answer:

- **RQ1:** How is the performance when applying RUBICK on real world fuzzing projects?
- **RQ2:** How is the quality of the fuzz drivers generated by RUBICK compared with fuzz drivers generated by state-of-the-art techniques and manually written ones?
- **RQ3:** Are the fuzz drivers improved by addressing the three key challenges?
- **RQ4:** How are the false positives produced by the fuzz drivers of RUBICK and other existing methods?
- **RQ5:** Can the fuzz drivers generated by RUBICK help to find vulnerabilities in real world fuzzing scenarios?

Evaluation Targets We apply RUBICK on library targets which are top usage (used by other apps/programs) third-party libraries supporting both PC and Android platforms. The usage data is crawled from maven repository [20] and appbrain [21]. The evaluated targets are the top six ranked by the number of APIs (Appendix D). Tbl. 1 details the attack surfaces identified from the six popular Java libraries. Note that the libraries and attack surfaces have a many-to-many relationship. Multiple attack surfaces inside one library are separated by the input formats they accept. For example, for `apachepoi` and `apachetar`, their libraries provide APIs to parse 12 and 22 different types of input formats, which means that they have 12 and 22 attack surfaces respectively. For these libraries, we pick the attack surface which accepts a popular input format.

Experiment Setup To fuzz Java programs, we use `jazzer` [22], which is a libfuzzer-based fuzzer used by OSS-FUZZ and ClusterFuzz. Following the suggestions from [23], all the evaluated fuzz drivers share the same input seeds, machine, and fuzzer options (`-jvm_args="-Xmx2048m" -close_fd_mask=3 -timeout=60 -rss_limit_mb=10240`). For fairness, the coverage of the fuzz driver itself is excluded, i.e., the comparison only covers the edge coverage of the target attack surfaces. All data used in evaluation are collected from 24 hours, 10 times repeated fuzzing results. In the plots, lines are average values and the shadows around the lines represent 95% confidence intervals. All the listed unique bugs including the false positives are manually deduplicated. First we group bugs based on their full stacks, then manually merge the groups whose root cause stacks have the same code location (function & line). The experiments are conducted on a Linux server with two Intel(R) Xeon(R) Gold 6248 CPU @ 2.50GHz processors and 188GB RAM.

4.1 Fuzz Driver Generation

To apply RUBICK on these attack surfaces, we built a crawler to collect the consumer programs. The crawler first locates the open-source consumer programs, then retrieves their jars as RUBICK's input. Specifically, a project is a consumer program if its code contains the package path of the attack surface, e.g., `com.github.junrar`. We use `src`, a CLI tool of Sourcegraph [24], to launch the match query among all open-source projects of Github, Gitlab, Bitbucket, etc. Our website [5] lists the used search patterns. For the matched consumer programs, we use heuristics to automatically retrieve their jars: ❶ find latest released jars from their webpages; ❷ try to build jars using common building commands, e.g., `mvn package`. As shown in Tbl. 2, we collect dozens to hundreds consumer programs for each attack surface. The second column shows the number of usable/matched consumers. A consumer is usable

Table 2: Statistics of Intermediate Results of RUBICK.

Attack Surface	**Crawling**		**Learning Inputs Preparation**					**Usage Automaton Learning**									**Fuzz Driver Synthesis**		
	# of Projs	# of Jars	# of Entries	# of Raw NFAs	# of APIs	# of Data Deps	CPU Sec (Pct.)	# of C_C	# of C_{MG}	# of E_{API}	# of E_{Mock}	# of E_{Cond}	# of E_{Total}	# of State	# of Tran	CPU Sec (Pct.)	# of Ctrl Flow	# of Data Flow	CPU Sec
`apachetar`	36/911	39/92	34,905	91	33	2,089	1,647 (**66%**)	0	0	22/33	0/1	16/54	38/88	165	223	832 (34%)	319	319	< 1
`apachepoi`	40/984	74/1,197	26,534	247	243	6,619	656 (34%)	1	3	69/243	8/15	12/109	89/367	89	94	1,289 (**66%**)	20	20	< 1
`itextpdf`	25/89	33/44	14,632	2,236	311	8,560	194 (3%)	1	3	75/311	59/88	78/365	212/764	308	348	7,223 (**97%**)	16	131,088	< 1
`junrar`	16/72	24/441	9,737	143	147	1,023	114 (16%)	0	0	49/147	0/0	14/52	63/199	120	150	617 (**84%**)	12	13	< 1
`pdfbox`	34/326	138/4,835	83,481	455	339	13,680	1,260 (29%)	0	2	54/339	9/14	36/184	99/537	127	148	3,127 (**71%**)	21	21	< 1
`zip4j`	62/514	28/262	9,175	41	49	1,635	298 (**69%**)	0	1	34/49	1/2	14/22	49/73	65	75	132 (31%)	5	5	< 1

if its jars are retrieved. The third column shows the number of usable/retrieved jars. A jar is usable if it contains the usage code and can be analyzed by `soot` [25].

First, RUBICK prepares the learning inputs from usable jars. For every function inside the consumers, RUBICK applies Raw NFA extraction. In 4th column of Tbl. 2, RUBICK analyzed more than 9,000 functions for every attack surface. The fifth column shows the amount of extracted non-empty Raw NFAs. The amounts of the contained API functions and the data dependencies are listed in the next two columns. The reason RUBICK *can practically analyze large amount of real world projects* is two-fold: ❶ The time complexity of its extraction algorithm is $O(E)$ where E is the amount of edges of the traversed ICFG; ❷ RUBICK configures `soot` to build a partial ICFG. Before ICFG construction, the classes of third party libraries were excluded using `SootClass.setPhantomClass`.

Next, RUBICK learns one usage automaton from the inputs. It first unifies the alphabet for all extracted Raw NFAs, then it distills all the Raw NFAs and merges them as the final usage automaton. The ninth and tenth columns show the number of the conflicts in unifying condition events C_C and mock API related event groups C_{MG}. The eleventh to fourteenth columns list the number of API events E_{API}, mock API events E_{Mock}, condition events E_{Cond}, and total events E_{Total} used in all automatons after/before distillation. The distillation decreases the number of the events since it removes invalid event sequences. Lastly, RUBICK synthesizes the fuzz driver. The last three columns show its detailed statistics.

4.2 Performance Assessment (RQ1)

Overall Performance In Tbl. 2, the last column in each component lists the cost of that component in CPU second and the percentage $\frac{cost}{total\ cost}$. The percentages of Fuzz Driver Synthesis are ignored since most values are less than 1‰. *Overall,* RUBICK *can generate fuzz drivers for real world projects with a reasonable time cost.* For large targets like `pdfbox` (~2700 APIs, ~100 consumers), RUBICK generates its fuzz driver in 2 CPU Hours. Besides, the Automata Learning component costs most cpu resources (63.88% on average). Note that the cost of Learning Input Preparation varies according to the amount and complexity of the entry functions inside the consumers. And the cost of Automata Learning increases exponentially when its lookahead value increases (see paragraph **Parameter Selection in Distillation**). Therefore, cost comparison

Lower Whisker	1st Quartile	Median	3rd Quartile	Upper Whisker
2.74	6.11	6.66	9.02	13.38

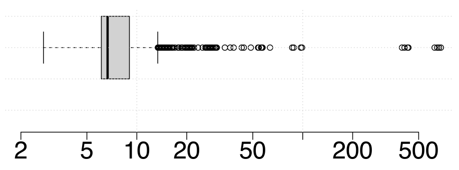


Figure 4: Boxplot and Statistics of Distillation Time Per Raw NFA. X-axis is the time in seconds. The axis is in log scale.

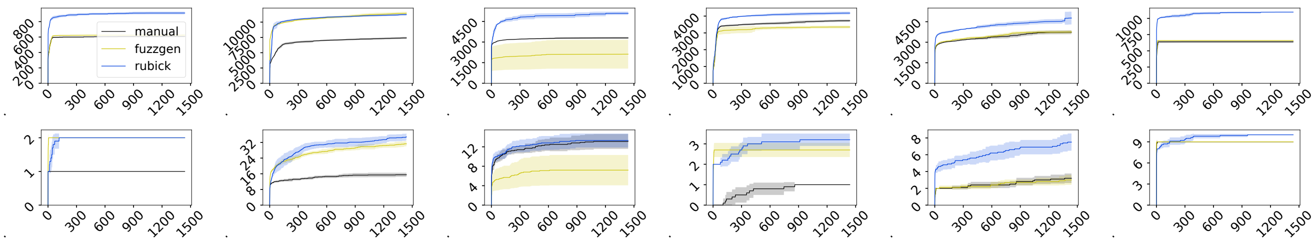
results can vary under different settings. However, mostly the Automata Learning component will have highest costs.

Performance of Usage Distillation To understand the performance of distillation, we did statistics for the distillation time of each automaton. Fig. 4 shows the box plot and the statistics. According to the 3rd Quartile (9.02) and Upper Whisker value (13.38), we conclude that *mostly distilling an automaton costs less than 14 CPU seconds.*

Parameter Selection in Distillation The only adjustable parameter in L* is the lookahead value (abbr as L) of the wp-method. We study the effects of L by comparing the learning cost and the learned automaton under different L. Tbl. 4 shows the total time/number of MQ when using four different L. Note that EQ is also counted since EQ intrinsically is using a set of MQ to find the counterexample. The results show that the cost increases exponentially when L increases. By comparing the learned automata, we found that: ❶ In most cases (99.47%, 3196/3213), all settings can learn the same automaton; ❷ In the rest 17 cases, L = 3 and L = 4 learn the same automaton. Compared with L = 4, L = 2 learns 8 automatons differently and L = 1 learns 17 automatons differently. By analyzing these 17 cases, we found that lower L increase the possibility of missing correct usage and summarizing false usage. Specifically, when L = m, the algorithm can falsely summarize the m + 1 repeated call sequences as a loop. Besides, it also misses the following correct call sequences. *Considering the high cost and minor learning outcome improvement of using a high L, we conclude that both 2 and 3 are suitable values in practice and we use 2 as the default value.*

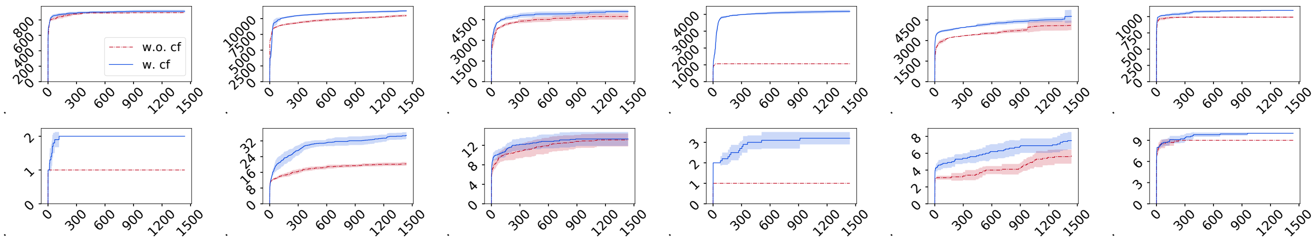
4.3 State-of-the-Art Comparison (RQ2)

Baselines We evaluated the effectiveness of RUBICK by comparing the fuzzing performance of its fuzz drivers with



(a) `apachetar` (b) `apachepoi` (c) `itextpdf` (d) `junrar` (e) `pdfbox` (f) `zip4j`

Figure 5: Coverage/Unique Bug (1st/2nd row) Per Time Comparisons for RQ2. X-, Y-axis are time (sec) and edge coverage/# of unique bugs.



(a) `apachetar` (b) `apachepoi` (c) `itextpdf` (d) `junrar` (e) `pdfbox` (f) `zip4j`

Figure 6: Coverage/Unique Bug (1st/2nd row) Per Time Comparisons for RQ3 C1. X-, Y-axis are time (sec) and edge coverage/# of unique bugs.

Table 3: Comparison on Metrics of Covered APIs (APIs) and API Sequence Cyclomatic Complexity (CC).

Attack Surface	`apachetar`		`apachepoi`		`itextpdf`		`junrar`		`pdfbox`		`zip4j`	
	APIs	CC	APIs	CC	APIs	CC	APIs	CC	APIs	CC	APIs	CC
manual	2	1	12	2	2	1	3	2	9	2	3	3
fuzzgen	23	1	56	1	55	1	18	1	21	1	24	1
rubick	22	60	69	7	54	23	49	32	75	42	34	12
w.o. cf	16	1	15	1	47	1	42	1	46	1	12	1
raw	28	224	217	435	307	939	116	169	210	1313	47	27

Table 4: Statistics of Learning Under Different Lookahead Values.

Metric	L = 1	L = 2	L = 3	L = 4
Time (CPU Sec)	1.21E+04	1.74E+04	7.89E+04	1.65E+06
# of Queries	1.22E+07	1.38E+08	2.86E+09	1.14E+11

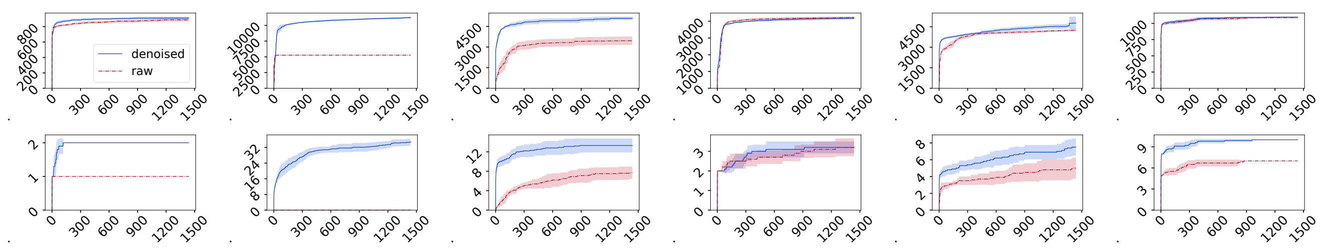
the drivers from FUZZGEN (abbr as `fuzzgen`) and manually written fuzz drivers (abbr as `manual`). Originally, FUZZGEN is written for C/C++ programs. We developed its Java version. We strictly followed the algorithms discussed in its paper and aligned the implementation detail with its released source code. Besides, since the fuzz drivers generated by FUZZGEN are too complex to be fuzzed when there are hundreds or thousands of consumer programs(as discussed in its paper). We manually filtered out the consumers with invalid usage and select the top five consumers ranked by including the most unique API calls. The filtered consumers contain usages which crash even under a valid input, e.g., the usage which misses necessary data dependencies for calling an API. If these consumers are used for FUZZGEN, its fuzz driver will always crash since the driver tries to execute all usages in every execution. The manually written fuzz drivers of `apachetar`, `pdfbox` are collected from OSS-Fuzz. For the rest, we invited a human expert who is not the author of this paper to manually write them. The expert is familiar with fuzzing and coding in Java but has no a priori knowledge of these attack surfaces. Writing these four fuzz drivers costs the expert around one week.

Fig. 5 shows the comparison results in metrics of edge coverage and unique bugs. The blue, yellow, and black lines stand for the coverage of `rubick`, `fuzzgen`, and `manual`, respectively. In both metrics, `rubick` shows apparent performance advantage over the other two baselines. Specifically, almost all p-values of `rubick` − `fuzzgen` and `rubick` − `manual` are smaller than 5.00e-2, which shows the statistical significance (see full figures in Appendix E Tbl. 8). Tbl. 3 lists the comparison on three more metrics. Results show that RUBICK mostly finds more unique bugs, covers more APIs, and has a higher API sequence complexity. *Overall, we conclude that* RUBICK *can generate more effective fuzz drivers than existing methods.*

4.4 Ablation Study (RQ3)

4.4.1 API Control Dependencies (C1)

To study the effectiveness of the API control dependency, we compare the fuzzing performance of the fuzz drivers with control flow sensitivity (abbr as `w. cf`) and without control flow sensitivity (abbr as `w.o. cf`). Both fuzz drivers are generated using RUBICK except that the condition events and mock API events are not identified for `w.o. cf`. Fig. 6 shows the comparison on the metrics of edge coverage and unique bugs. Blue solid line and red dotted line are `w. cf` and `w.o. cf` respec-



(a) `apachetar` (b) `apachepoi` (c) `itextpdf` (d) `junrar` (e) `pdfbox` (f) `zip4j`

Figure 7: Coverage/Unique Bug (1st/2nd row) Per Time Comparisons for RQ3 C2. X-, Y-axis are time (sec) and edge coverage/# of unique bugs.

Table 5: No. of Independent Usage Scenarios w. & w.o. Denoise

	apachetar	apachepoi	itextpdf	junrar	pdfbox	zip4j
raw	1,099	22,625	17,713	222	7,861	24
denoised	319	20	16	12	21	5
R (1 - $\frac{denoised}{raw}$)	70.97%	99.91%	99.91%	94.59%	99.73%	79.17%

tively. `w. cf` shows clear advantage in the plots. Almost all p-values of `w. cf` − `w.o. cf` are smaller than 5.00e-2, which shows the statistical significance (see Tbl. 8). Besides, by comparing the `rubick` (`w.cf`) and `w.o. cf` in Tbl. 3, we found that control flow sensitivity also helps the fuzz driver to carry more diverse usages by covering more APIs. Appendix F further provides a case study revealing how control-flow-sensitivity helps for more efficient bug hunting. *The evaluation shows that considering control flow sensitivity can improve both the quality and performance of the fuzz drivers.*

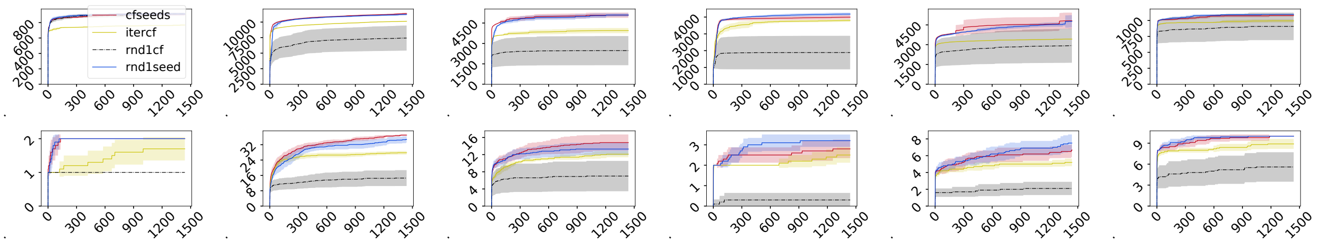
4.4.2 Denoise (C2)

To study the effectiveness of the denoise techniques, we compare the fuzz drivers generated with the usage distillation techniques (abbr as `denoised`) and without these techniques (abbr as `raw`). Specifically, the process of generating `raw` skips the techniques described in Section 3.3.2. Tbl. 5 lists the number of the independent usage scenarios for `denoised` and `raw`. Apparently, `raw` has a significantly more complex usage automaton than `denoised`. However, most of the complexity is unnecessary since a majority of the usage inside `raw` are noises (redundant or invalid event sequences). As shown in the Tbl. 5, RUBICK filters out 70.97% to 99.91% usage scenarios. Fig. 7 lists the comparison in metrics of edge coverage and unique bugs. The blue solid line and red dotted line represent `denoised` and `raw` respectively. In most cases, `denoised` reaches higher coverage and finds more unique bugs with statistical significance (see p-values in Tbl. 8). Interestingly, as shown in Tbl. 3, the `raw` covers more APIs and has a notably higher API sequence complexity. This further proves that the unnecessary complexity inside the `raw` does not benefit but hurt its fuzzing performance. *Usage distillation is a necessary step, which improves the overall fuzzing performance by significantly reducing the complexity of usage automaton.*

4.4.3 Automata-Guided Fuzz Driver (C3)

Baselines In RUBICK's fuzz driver, the fuzzer can change the testing usage scenario by changing the value of leading bytes of the input. To understand the effectiveness of the scheduling interface, we compared the fuzz drivers with the following four settings. ❶ `rnd1seed`, the default strategy used by RUBICK. The fuzzing is started with one initial input seed whose leading bytes are assigned with random values. In other words, the fuzzer has a randomly picked testing scenario as the startup. For repeated experiments, each fuzzer instance will have its own starting scenario. ❷ `cfseeds`, it is same as `rnd1seed` except that there are multiple initial seeds where each seed picks a different scenario. In other words, `cfseeds` gives fuzzer full information for its scenario scheduling at the beginning of the fuzzing. ❸ `rnd1cf`, a setting that the schedule interface is disabled. Before fuzzing, it will be randomly bound with one usage scenario and the fuzzer can only fuzz that scenario throughout the 24h experiment. In repeated experiments, the scenarios among the fuzzing instances are picked and bound independently. `rnd1cf` stands for the normal fuzzing strategy without scenario scheduling. ❹ `itercf`, a setting that the schedule interface is also disabled. However, different from `rnd1cf`, it iterates all usage scenarios in each single fuzzing iteration. `rnd1cf` represents the strategy that every usage scenario is equally scheduled.

Fig. 8 shows the comparison results of four settings on metrics of edge coverage and unique bug: ❶ In all attack surfaces, `rnd1seed` and `cfseeds` are higher than `itercf` and `rnd1cf` in both two metrics with statistical significance (see p-value detail in Tbl. 8). This shows the scheduling interface is effective. ❷ Overall, it is hard to pick the dominant setting between `cfseeds` and `rnd1seed`. For coverage, `cfseeds` has higher initial coverage than `rnd1seed` in `apachepoi` and `itextpdf`, which is reasonable since `cfseeds` has more initial seeds. However, the final performances of them are similar. For unique bugs, `cfseeds` performs better than `rnd1seed` in `apachepoi` and `itextpdf` but worse in `pdfbox` and `junrar`. An explanation is that the exploration ability of the fuzzer itself is good enough, therefore the fuzzer can quickly explore all usage scenarios. ❸ Mostly, the confidence intervals (shadows along the lines) of `rnd1cf` are observably wider than others.



(a) apachetar (b) apachepoi (c) itextpdf (d) junrar (e) pdfbox (f) zip4j

Figure 8: Coverage/Unique Bug (1st/2nd row) Per Time Comparisons for RQ3 C3. X-, Y-axis are time (sec) and edge coverage/# of unique bugs.

Table 6: Unique False Positives Statistics. **Bad Data Dep**, **Invalid Call Seq**, and **Improper Cond** stand for the false crashes caused by missing or invalid API data dependencies, invalid API call sequences, and missing API control dependencies, respectively. The number of false dead loops are counted into a standalone type **Dead Loop**.

	`fuzzgen`	`rubick`	`w.o. cf`	`raw`
Bad Data Dep	7	0	0	4
Invalid Call Seq	2	0	0	0
Improper Cond	22	7	10	2
Dead Loop	0	0	0	2
Total	31	7	10	8

This indicates that the performance of `rnd1cf` among multiple rounds of fuzzing is less stable. Interestingly, this reveals that the potentials of different usage scenarios are different; therefore, a scheduling interface for maximizing the utilization of multiple usages is necessary. Besides, the execution speed of `itercf` is many times slower than others since it executes all usage scenarios during one fuzzing iteration. *In summary, the scheduling interface provided by* RUBICK *can significantly improve the utilization of the multiple usage scenarios, and both* `rnd1seed` *and* `cfseeds` *are suitable setups.*

4.5 False Positives (RQ4)

To study the false positives produced by the fuzz drivers, we manually identified and deduplicated false positives from all evaluated fuzz drivers. Tbl. 6 provides the FP statistics (Tbl. 9 in Appendix G details the FP rate statistics.) Note that manually written fuzz drivers are not listed since they have not produced any false positives. `rubick` has the least number of false positives. The root cause of the false positives from `rubick` is that it has learned unsound usage from the consumers. Besides, the rest of the false positives from other fuzz drivers can either be avoided by RUBICK's control flow modeling or be filtered out by RUBICK's distillation. *Generally,* RUBICK *produces less false positives than* FUZZGEN. *And the control flow sensitivity and denoising helps in reducing the false positives caused by fuzz drivers.*

All of `rubick`'s false positives have the same root cause. They are of type **Improper Cond** and appear in `itextpdf`. Typically, a consumer can contain an unsound usage which calls `page.getxxx()` without checking whether the `page` is `null` or not. The distillation in `rubick` cannot filter this unsound usage out since this usage works well with a valid pdf file. Except this, the rest false positives found in other fuzz drivers can be filtered or avoided by `rubick`: ❶ For **Bad Data Dep**, `fuzzgen` passes fixed values which are supposed to be loop control variables for some APIs. For example, `fuzzgen` uses `getSheet(0)` while the correct usage is `getSheet(i)` where `i` ranges from `0` to the number of sheets in this input. When the mutated input contains zero pages, the fuzz driver will crash. `raw` has false positives since it calls an API with missing arguments. Indeed it does not have enough data dependencies to fill in all arguments of that API. ❷ For **Invalid Call Seq**, `fuzzgen` calls `pdf.getNumberOfPages()` after it calls `pdf.close()`. This invalid sequence does not come from the consumer but is introduced by `fuzzgen`'s coalescing strategy. ❸ For **Improper Cond**, `fuzzgen` and `w.o. cf` call APIs without checking the API usage precondition. For example, they call `iteratorX.next()` without checking `iteratorX.hasNext() == true`, which causes false crashes. ❹ For **Dead Loop**, `raw` got stuck since it does not filter out dead loops inside its automata. All the cases can be solved with `rubick`'s control flow modeling and distillation features.

4.6 Real-world Application (RQ5)

Fuzz Driver Generation RUBICK has been used to generate fuzz drivers for eleven attack surfaces, including the attack surfaces in evaluated projects, `metadata-extractor`, `Apache Tika`, `fastjson`, and `jackson`. The selection criteria is that they are popular Java projects on both PC and Android systems. Therefore, finding the bugs of these projects can benefit the software of both systems. Additionally, we directly generated human readable fuzz drivers containing one usage scenario. Two of them have been merged into OSS-FUZZ.

Long-term Fuzzing Campaign With the generated fuzz drivers, we conducted a three-month fuzzing campaign. In total, 199 bugs have been found and responsibly reported to the vendors (see full list in Appendix H). The types of the found bugs cover Uncaught Exception, Stack Overflow, Out-Of-Memory, and Infinite Loop. Four CVE numbers have been assigned. To further evaluate the impact of these bugs, we built a workflow to semi-automatically identify the APPs which are truely exploitable by these bugs. We successfully exploited eleven APPs and four of them have 10,000,000+ download counts in Google Store. Specifically, the APPs can behave abnormally (crash, stuck, etc) when users try to open the POCs. One interesting case is that a top APP (10,000,000+ download count) will be stuck and occupy all memory as long as the POC exists in the file system of the phone and the user refreshes the file list in UI. Simply restarting the APP cannot solve this issue since the APP tries to recover the last opened file list in the background. We've responsibly notified all APP vendors about the security issues and provided fix suggestions. More details can be found in our website [5].

5 Limitation & Future Work

More Types of API Dependencies Currently, RUBICK only extracts common API control dependencies (branches and loops related with outputs of API functions) from the consumers. However, there exists API dependencies that cannot be precisely described by RUBICK. For example, the input value of the API functions may also affect its following usages, or some usages can be out of the description ability of DFA such as time related usage. To fully support these dependencies, extra modeling or even new models need to be developed. We leave the support as future work.

More Learning Sources The two major learning sources of RUBICK are the static analysis of the example code and the execution information of the generated code. On one hand, these sources provide necessary information for generating valid fuzz drivers. On the other hand, the learning ability of RUBICK is limited by them. For instance, due to the lack of proper semantic guidance, it is hard to reasonably infer usage of the APIs which are not used in the consumer programs from the used. Domain knowledge, documentations, code comments, or even function names are complementary sources containing high level API semantics which are promising to improve the learned usage. In the future, we plan to research learning usage from multiple dimensions of sources.

Other Programming Languages Currently, RUBICK generates fuzz drivers for Java code. There are no fundamental differences for applying RUBICK to other languages since its core methods are based on general concepts of modern languages. However, besides engineering efforts for adaptation, extending RUBICK to other languages may require additional efforts to solve language-specific challenges. For example, for languages requiring standalone compilation, collecting analyzable consumer programs requires language-specific heuristics for compilation. For generating C/C++ fuzz drivers, the data dependency extraction in RUBICK may need additional strategies for handling pointers and structs.

6 Related Work

Fuzz Driver Generation Recently, several works have been published on this emerging research topic. FUDGE [26] synthesizes candidate fuzz drivers based on the code snippets sliced from the consumer programs. After that, human experts need to refine the drivers and determine which driver should be used. Comparatively, FUZZGEN [1] is more automatic. It learns an Abstract API Dependency Graph (A^2DG) which contains the call sequences of the APIs leveraging the data flow information. By traversing the A^2DG, it synthesizes the fuzz drivers. INTELLIGEN [3] proposes metrics to rank the API functions and synthesizes fuzz drivers for APIs with higher ranking. GRAPHFUZZ [27] is a semi-automatic tool which generates fuzz drivers from mutated object lifetime-aware data flow graphs to test the target libraries. WINNIE [28] and APICRAFT [2] focus on solving the challenges of generating fuzz drivers for closed-source targets. WINNIE synthesizes the fuzz driver from traces and develops a fast cloning technique to boost the fuzzer in Windows systems. Comparatively, APICRAFT focuses more on finding better combinations of the API data dependencies. Among them, FUZZGEN is the most related work to RUBICK since both of them aim to automatically generate fuzz drivers via static analysis on the consumer programs. However, RUBICK differs from FUZZGEN by focusing on solving the discussed three challenges (Section 2.2).

API Usage and Specification Mining One popular way to learn the API specification is through dynamic analysis [29, 30]. This approach learns finite state machines (FSMs) which describe legal sequences of method calls from large method traces. Besides, some works mine the usage or API properties to check the misuse or unsafe use of the APIs [31–34]. These works do not focus on fuzz driver generation, but they provide inspirations for the current as well as future design of RUBICK.

Advanced Fuzzing Techniques Fuzzing is one of the most practical techniques for detecting zero-day software vulnerabilities [35]. Many techniques on improving fuzzing have been published in recent years [36–52]. These techniques are orthogonal to RUBICK since the fuzz drivers generated by RUBICK can be supplied to any fuzzer.

7 Conclusion

In this paper, we propose RUBICK, an automata based fuzz driver generation technique. Comparing with existing works, RUBICK has three key merits: ❶ it embeds more diverse API usages such as branches and loops; ❷ it learns API usages from large-scale real world projects; ❸ it improves the utilization of multiple usage scenarios. In evaluation, RUBICK shows great advantage over the baselines. RUBICK has been used to generate fuzz drivers for 11 popular Java libraries and two of them have been merged into OSS-Fuzz. In total, RUBICK has discovered 199 new bugs, including four CVEs, affecting popular PC and Android software (10,000,000+ downloads).

8 Acknowledgments

We thank our anonymous shepherd and reviewers for their insightful comments on our work, thank Dr. Li Yeting for his valuable suggestions related with automata learning, and thank Mr. Huang Wenjie for his helps on real world application of RUBICK. This research is supported by the National Research Foundation, Singapore under its the AI Singapore Programme (AISG2-RP-2020-019), the National Research Foundation through its National Satellite of Excellence in Trustworthy Software Systems (NSOE-TSS) project under the National Cybersecurity R&D (NCR) Grant award no. NRF2018NCR-NSOE003-0001, the RIE2020 Industry Alignment Fund –Industry Collaboration Projects (IAF-ICP) Funding Initiative, as well as cash and in-kind contribution from the industry partner(s), the Hong Kong RGC Project (No. PolyU15222320) and HKPolyU Grant (ZVG0), and the National Natural Science Foundation of China (Grant Nos. 62125205).

References

[1] K. Ispoglou, D. Austin, V. Mohan, and M. Payer. Fuzzgen: Automatic fuzzer generation. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*.

[2] C. Zhang, X. Lin, Y. Li, Y. Xue, J. Xie, H. Chen, X. Ying, J. Wang, and Y. Liu. Apicraft: Fuzz driver generation for closed-source sdk libraries. In *30th USENIX Security Symposium (USENIX Security 21)*.

[3] M. Zhang, J. Liu, F. Ma, H. Zhang, and Y. Jiang. Intelligen: automatic driver synthesis for fuzz testing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*.

[4] K. Serebryany. {OSS-Fuzz}-google's continuous fuzzing service for open source software.

[5] Website of RUBICK. https://sites.google.com/view/rubick-lore/home.

[6] D. Angluin. Learning regular sets from queries and counterexamples. *Information and computation*.

[7] Human expert's confusion on writing fuzz drivers in oss-fuzz. https://bit.ly/3tdZT8k.

[8] Regular expression syntax of python re. https://bit.ly/3MeAulT.

[9] J. E. Hopcroft, R. Motwani, and J. D. Ullman. Introduction to automata theory, languages, and computation. *Acm Sigact News*.

[10] F. B. Khendek, S. Fujiwara, G. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Transactions on software engineering*.

[11] J. Hopcroft. An n log n algorithm for minimizing states in a finite automaton. In *Theory of machines and computations*.

[12] Apache commons compress project. https://bit.ly/3NnNzuo.

[13] Apache poi - the java api for microsoft documents. https://bit.ly/3x4QiBX.

[14] itext 7 - a java pdf solution. https://bit.ly/3az3CH7.

[15] junrar - java rar library. https://bit.ly/3Q17fpv.

[16] Apache pdfbox - a java pdf library. https://bit.ly/3GQtf2v.

[17] zip4j - a java zip libraries. https://bit.ly/3zn8nOF.

[18] M. Isberner, F. Howar, and B. Steffen. The open-source learnlib. In *Computer Aided Verification*.

[19] z3py – z3 api in python. https://bit.ly/3O1DKlP.

[20] Maven repository. https://bit.ly/3RU2EGe.

[21] Appbrain. https://bit.ly/3S8mI7y.

[22] Coverage-guided, in-process fuzzing for the jvm. https://bit.ly/3anbZWg.

[23] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*.

[24] Sourcegraph. https://bit.ly/3mn2yJj.

[25] Soot. https://bit.ly/3MgKI55.

[26] D. Babić, S. Bucur, Y. Chen, F. Ivančić, T. King, M. Kusano, C. Lemieux, L. Szekeres, and W. Wang. Fudge: fuzz driver generation at scale. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[27] H. Green and T. Avgerinos. Graphfuzz: Library api fuzzing with lifetime-aware dataflow graphs. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering*.

[28] J. Jung, S. Tong, H. Hu, J. Lim, Y. Jin, and T. Kim. Winnie: Fuzzing windows applications with harness synthesis and fast cloning.

[29] M. Pradel and T. R. Gross. Automatic generation of object usage specifications from large method traces. In *2009 IEEE/ACM International Conference on Automated Software Engineering*.

[30] M. Pradel and T. R. Gross. Leveraging test generation and specification mining for automated bug detection without false positives. In *2012 34th International Conference on Software Engineering (ICSE)*.

[31] I. Yun, C. Min, X. Si, Y. Jang, T. Kim, and M. Naik. {APISan}: Sanitizing {API} usages through semantic {Cross-Checking}. In *25th USENIX Security Symposium (USENIX Security 16)*.

[32] B. He, V. Rastogi, Y. Cao, Y. Chen, V. Venkatakrishnan, R. Yang, and Z. Zhang. Vetting ssl usage in applications with sslint. In *2015 IEEE Symposium on Security and Privacy*.

[33] H. Chen and D. Wagner. Mops: an infrastructure for examining security properties of software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*.

[34] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*.

[35] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities.

[36] M. Böhme, V. Pham, and A. Roychoudhury. Coverage-based greybox fuzzing as markov chain.

[37] Y. Li, B. Chen, M. Chandramohan, S. Lin, Y. Liu, and A. Tiu. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*.

[38] P. Chen and H. Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*.

[39] H. Chen, S. Guo, Y. Xue, Y. Sui, C. Zhang, Y. Li, H. Wang, and Y. Liu. {MUZZ}: Thread-aware greybox fuzzing for effective bug hunting in multithreaded programs. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*.

[40] Y. Li, Y. Xue, H. Chen, X. Wu, C. Zhang, X. Xie, H. Wang, and Y. Liu. Cerebro: context-aware adaptive fuzzing for effective vulnerability detection. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[41] Y. Chen, P. Li, J. Xu, S. Guo, R. Zhou, Y. Zhang, T. Wei, and L. Lu. Savior: Towards bug-driven hybrid testing. In *2020 IEEE Symposium on Security and Privacy (SP)*.

[42] C. Lyu, S. Ji, C. Zhang, Y. Li, W. Lee, Y. Song, and R. Beyah. Mopt: Optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*.

[43] T. Yue, P. Wang, Y. Tang, E. Wang, B. Yu, K. Lu, and X. Zhou. Ecofuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit. In *29th USENIX Security Symposium (USENIX Security 20)*.

[44] C. Zhang, Y. Li, H. Chen, X. Luo, M. Li, A. Q. Nguyen, and Y. Liu. Biff: Practical binary fuzzing framework for programs of iot and mobile devices. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

[45] J. Choi, D. Kim, S. Kim, G. Grieco, A. Groce, and S. K. 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)*.

[46] J. Choi, K. Kim, D. Lee, and S. K. Cha. Ntfuzz: Enabling type-aware kernel fuzzing on windows with static binary analysis. In *2021 IEEE Symposium on Security and Privacy (SP)*.

[47] M. Böhme, V. J. Manès, and S. K. Cha. Boosting fuzzer efficiency: An information theoretic perspective. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[48] Y. Zheng, Y. Li, C. Zhang, H. Zhu, Y. Liu, and L. Sun. Efficient greybox fuzzing of applications in linux-based iot devices via enhanced user-mode emulation. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*.

[49] Y. Aafer, W. You, Y. Sun, Y. Shi, X. Zhang, and H. Yin. Android {SmartTVs} vulnerability discovery via {Log-Guided} fuzzing. In *30th USENIX Security Symposium (USENIX Security 21)*.

[50] T. Cloosters, J. Willbold, T. Holz, and L. Davi. Sgxfuzz: Efficiently synthesizing nested structures for sgx enclave fuzzing. In *USENIX Security*.

[51] S. Schumilo, C. Aschermann, A. Jemmett, A. Abbasi, and T. Holz. Nyx-net: network fuzzing with incremental snapshots. In *Proceedings of the Seventeenth European Conference on Computer Systems*.

Algorithm 3 Solution Set Division

```
Input: $L_I$ (List of solution sets $\{R_1, R_2 ... R_m\}$)
Output: $L_O$ (List of solution sets $\{R'_1, R'_2 ... R'_n\}$)
 1: procedure SOLUTION-SET-DIVISION($L_I$)
 2:     $L_O \leftarrow L_I$
 3:     while $\exists\ R'_i, R'_j \in L_O, R'_i \cap R'_j \neq \emptyset$
 4:         Eliminate $\emptyset$ inside $L_O$
 5:         Remove duplicate $R'$ inside $L_O$
 6:         for $k$ in $[len(L_O), len(L_O) - 1 ... 2]$
 7:             if $\exists\ R'_{x1}, R'_{x2} ... R'_{xk} \in L_O, R'_{x1} \cap R'_{x2} ... \cap R'_{xk} \neq \emptyset$
 8:                 Remove $R'_{x1}, R'_{x2} ... R'_{xk}$ from $L_O$
 9:                 $R'_{new} \leftarrow R'_{x1} \cap R'_{x2} ... \cap R'_{xk}$
10:                 $L_O \overset{+}{\leftarrow} R'_{new}, (R'_{x1} - R'_{new}), (R'_{x2} - R'_{new}) ... (R'_{xk} - R'_{new})$
11:                 break
12:     return $L_O$
13: end procedure
```

[52] Q. Liu, C. Zhang, L. Ma, M. Jiang, Y. Zhou, L. Wu, W. Shen, X. Luo, Y. Liu, and K. Ren. Firmguide: Boosting the capability of rehosting embedded linux kernels through model-guided kernel execution. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

[53] Multiplication principle. https://bit.ly/3MjY6W7.

[54] N. Nachar et al. The mann-whitney u: A test for assessing whether two independent samples come from the same distribution.

A Algorithm for Unifying Condition Letters

To solve the conflict of two sets of condition letters, RUBICK needs to find a set of new condition letters which can represent both two sets. For example, given two sets of conflicting condition letters $A_1 : v < 1, A_2 : v \geq 1$ and $B_1 : v < 2, B_2 : v \geq 2$, the new set can be $C_1 : v < 1, C_2 : 1 \leq v < 2, C_3 : v \geq 2$ where $A_1 \leftrightarrow C_1$, $A_2 \leftrightarrow C_2 || C_3$, $B_1 \leftrightarrow C_1 || C_2$, $B_2 \leftrightarrow C_3$.

Note that each condition letter intrinsically represents a solution set which satisfies the condition. And the operations of the conditions can be mapped to the operations of the solution sets. For example, given two conditions C_i, C_j and their solution sets R_i, R_j, C_i && $C_j == true \leftrightarrow R_i \cap R_j$, $C_i == true$ && $C_j == false \leftrightarrow R_i - R_j, C_i \,||\, C_j == true \leftrightarrow R_i \cup R_j$.

The conflict of two sets of condition letters is caused by the intersection among two or more solution sets. Intuitively, by defining all intersected and disjoint parts among the solution sets as standalone condition letters, each original condition letter can be represented as their combinations. Algorithm 3 shows how RUBICK divides the solution set to a new set separating the intersected and disjointed parts. It accepts a list of all related solution sets as input L_I and outputs the divided solution sets. In each iteration (line 4 to line 11), it picks one intersection which has highest k (k is the number of the intersected solution sets). The rationale is that starting from picking an intersection with highest k can avoid missing the intersected solution sets. The algorithm can finally stop since it removes one intersection of the solution sets per iteration (line 4 to line 11) and the number of intersections is finite. Besides, any original solution set R_x where $R_x \in L_I$ can be represented as the union of solution sets from L_O. To represent R_x, we only need to find all subsets of R_x from L_O. Our website [5] posts a running example of this algorithm.

Algorithm 4 Independent Usage Scenarios Collection

```
Input: $A$ (Usage Automaton)
Output: $L_{cf}$ (Control Flow Usage List)
 1: procedure DFS(A, curS, curPath, curColorBox)
 2:     colorBoxSet $\leftarrow \emptyset$
 3:     numOfChoices, choices $\leftarrow$ Calc-Choices(A, curS)
 4:     if numOfChoices $== 0$
            ▷ reaches the end of automaton
 5:         colorBoxSet $\overset{+}{\leftarrow}$ curColorBox
 6:     else
 7:         dye $\leftarrow$ (numOfChoices $> 1$)
 8:         for choice $\in$ choices
 9:             curPath $\overset{+}{\leftarrow}$ curS
10:             if dye
11:                 curColorBox $\overset{+}{\leftarrow}$ Gen-Color(choice)
12:             subColorBoxSetList $\leftarrow$ []
13:             for event, nextS $\in$ choice
14:                 if nextS $\notin$ curPath
15:                     subColorBoxSetList $\overset{+}{\leftarrow}$ DFS(A, nextS, curPath, curColorBox)
16:                 else
                        ▷ meets the loop
17:                     colorBoxSet $\overset{+}{\leftarrow}$ curColorBox
18:             colorBoxSet $\overset{+}{\leftarrow}$ Crossover( subColorBoxSetList)
19:             if dye
20:                 curColorBox $\leftarrow$ curColorBox $-$ color
21:             curPath $\leftarrow$ curPath $-$ curS
22:     return colorBoxSet
23: end procedure
24: $L_{cf} \leftarrow$ []
25: for initialS $\in$ Get-Initial-States(A)
26:     colorBoxSet $\leftarrow$ DFS(A, initialS, [], $\emptyset$)
27:     $L_{cf} \overset{+}{\leftarrow}$ Convert-ColorBox-To-CF(colorBoxSet)
```

In implementation, the SAT solver is used to answer whether the condition represented by a solution set has an answer or not. For example, $R_i \cap R_j \neq \emptyset$ means SAT solver can find a solution satisfies C_i && $C_j == true$. The SAT solver provides base functionality for this algorithm, it is used in line 3, 4, 5, 7, used for simplifying the conditions represented by L_O, and for finding the subsets to represent the original solution set.

Table 7: Statistics of All Applied Projects of RUBICK.

Attack Surface	pdfbox	apachepoi	itextpdf	junrar	zip4j	apachetar	jackson	tika-jpeg	fastjson	java-json	metadata-jpeg
# of APIs	2779	1320	810	619	521	195	175	162	159	137	58

Table 8: P-values (Mann Whitney u test) for Evaluation. The p-value which means statistical significance (≤ 5e-2) is highlighted as bold.

		apachetar		apachepoi		itextpdf		junrar		pdfbox		zip4j	
		cov	bug	cov	bug	cov	bug	cov	bug	cov	bug	cov	bug
RQ1	rubick – fuzzgen	**8.73e-05**	-	1.06e-01	**3.34e-03**	**8.93e-05**	**3.00e-04**	**9.13e-05**	**1.61e-02**	**9.13e-05**	**7.83e-05**	**2.95e-05**	**7.97e-06**
	rubick – manual	**8.01e-05**	**7.97e-06**	**9.13e-05**	**8.54e-05**	**9.13e-05**	3.23e-01	**9.13e-05**	**1.64e-05**	**9.13e-05**	**7.69e-05**	**5.16e-05**	**7.97e-06**
RQ2 C1	w.o. cf – w. cf	6.59e-02	**7.97e-06**	**9.13e-05**	**8.63e-05**	**1.81e-03**	4.53e-01	**3.19e-05**	**1.64e-05**	**1.10e-03**	**6.02e-03**	**7.21e-05**	**7.97e-06**
RQ2 C2	denoised – raw	5.20e-02	**7.97e-06**	**7.43e-05**	**3.12e-05**	**9.08e-05**	**1.15e-04**	3.39e-01	4.81e-01	**1.41e-03**	**3.29e-03**	**1.11e-02**	**7.97e-06**
RQ2 C3	rnd1seed – cfseeds	4.70e-01	-	1.36e-01	**8.50e-03**	2.85e-01	1.25e-01	**1.29e-02**	6.12e-02	1.37e-01	8.77e-02	**3.23e-02**	-
	rnd1seed – itercf	**9.08e-05**	**3.84e-02**	**9.13e-05**	**7.83e-05**	**9.08e-05**	6.63e-02	**9.13e-05**	**4.08e-03**	**9.13e-05**	**7.81e-04**	**8.49e-05**	**2.71e-03**
	rnd1seed – rnd1cf	1.63e-01	**7.97e-06**	**9.13e-05**	**8.83e-05**	**9.13e-05**	**4.40e-04**	**2.09e-04**	**3.68e-05**	**1.64e-04**	**7.65e-05**	**8.54e-05**	**1.64e-05**
	cfseeds – itercf	**8.98e-05**	**3.84e-02**	**9.08e-05**	**4.74e-05**	**9.08e-05**	**8.83e-03**	**1.06e-02**	1.51e-01	**9.13e-05**	**3.21e-03**	**2.82e-04**	**2.71e-03**
	cfseeds – rnd1cf	3.11e-01	**7.97e-06**	**9.08e-05**	**5.43e-05**	**9.13e-05**	**7.97e-05**	**1.36e-03**	**5.02e-05**	**6.55e-04**	**7.60e-05**	**3.58e-03**	**1.64e-05**
	itercf – rnd1cf	**9.13e-05**	**8.08e-04**	**1.41e-03**	**7.83e-05**	**9.08e-05**	**3.61e-04**	**1.03e-02**	**4.99e-05**	7.02e-02	**6.39e-05**	8.06e-02	**4.52e-05**

B Algorithm for Independent Usage Scenario Collection

Algorithm 4 lists the details of collecting independent usage scenarios from a given usage automaton. Before discussing the algorithm, we explain some properties for counting the independent usage scenarios. Take the automaton in Fig. 1.e) as an example, only the branch in state 1 increases the number of independent usage scenarios. This is because events `B` and `H` are independent of each other. When the fuzz driver reaches state 1, it can decide to follow any branch by triggering either `B` or `H`, i.e., calling any of the two API functions. Comparatively, the branches of state 3 and 5 do not affect the number of independent usage scenarios since the events on all branches are related with each other, e.g., in state 3, fuzz driver can only trigger either `y` or `n` depending on the value of `ret_C`. Note that the case can be more complicated when both the function events and condition events appear in the branches of a given state. To correctly collect the usage scenarios, a base observation is that every independent usage scenario can be identified by the choices it made in these branches. Therefore, using the choice set as the signature, we can group all paths of the automaton. Each group is identified as an independent usage scenario. Algorithm 4 presents a DFS-based approach to this idea. Each choice is assigned with a unique color and every path has its own color box containing the colors it has been dyed (the choices it made along this path). The algorithm traverses the automaton starting from every initial state (line 25 – 27). For each state, it first calculates its independent choices. Zero choice means the end of the path (line 4). The colorbox will only be dyed when there are multiple choices (line 10 – 11). The algorithm iterates each transition of each choice to continue the DFS traverse. For the choice which has more than one transition (such as the `y` and `n` for state 3 in above example), the $colorBoxSet$ of each transition should be crossovered according to the Multiplication Principle [53] (line 18). Finally, it converts each color box into an usage scenario (line 27).

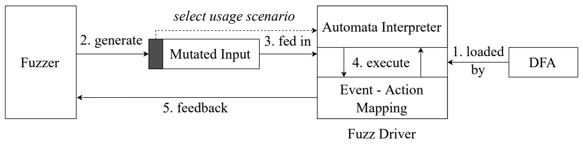


Figure 9: Workflow of Automata-Guided Fuzz Driver

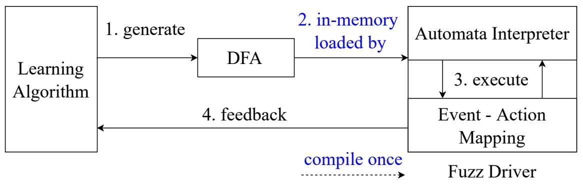


Figure 10: Workflow of Membership Query (MQ)

C Detail of Automata-Guided Fuzz Driver

General Workflow Fig. 9 illustrates the workflow of our fuzz driver. Step 1 is a one-time effort which is done at the beginning of the fuzzing. Step 2-5 represent one fuzzing iteration. Specifically, the italic sentence indicates that some bytes of the mutated input are used for specifying the testing target scenario, which is the designed scheduling interface.

Boosting Active Learning The automata-guided fuzz driver technique not only helps the scenario scheduling but also significantly boosts the usage distillation process. In distillation, RUBICK needs to dynamically validate the event sequences for answering MQ. Initially, RUBICK did this in three steps: ❶ conversion from usage to code; ❷ code compilation; ❸ code execution. RUBICK met performance issues since millions of compilations are unaffordable. RUBICK uses the

Table 9: Statistics of False Positive Rate. FP, Ttl stand for the amount of deduplicated false positives, total bugs respectively. FP-raw, Ttl-raw stand for the amount of their non-deduplication versions. Pct. stands for false positive rate whose value is either $\frac{FP}{Ttl}$ or $\frac{FP-raw}{Ttl-raw}$ according to its column. The field which has the smallest Pct. in that column is highlighted in bold.

Attack Surface	*apachetar*		*apachepoi*		*itextpdf*		*junrar*		*pdfbox*		*zip4j*	
	FP/Ttl (Pct.)	FP-raw/Ttl-raw (Pct.)	FP/Ttl (Pct.)	FP-raw/Ttl-raw (Pct.)	FP/Ttl (Pct.)	FP-raw/Ttl-raw (Pct.)	FP/Ttl (Pct.)	FP-raw/Ttl-raw (Pct.)	FP/Ttl (Pct.)	FP-raw/Ttl-raw (Pct.)	FP/Ttl (Pct.)	FP-raw/Ttl-raw (Pct.)
`fuzzgen`	16/18(88.89%)	160/77655(0.21%)	4/42(9.52%)	40/8458(0.47%)	3/12(25.00%)	30/3430(0.87%)	1/4(25.00%)	10/8446(0.12%)	1/9(11.11%)	10/37107(0.03%)	6/15(40.00%)	60/398(15.08%)
`rubick`	**0/2(0.00%)**	**0/2660(0.00%)**	**0/46(0.00%)**	**0/5124(0.00%)**	7/28(25.00%)	48/2123(2.26%)	**0/4(0.00%)**	**0/691(0.00%)**	**0/16(0.00%)**	**0/43039(0.00%)**	**0/10(0.00%)**	**0/338(0.00%)**
`wocf`	2/3(66.67%)	20/9960(0.20%)	**0/24(0.00%)**	**0/5916(0.00%)**	**3/20(15.00%)**	**43/4600(0.93%)**	3/5(60.00%)	30/40(75.00%)	**0/10(0.00%)**	**0/55105(0.00%)**	2/11(18.18%)	20/310(6.45%)
`raw`	3/4(75.00%)	8732/9137(95.57%)	2/2(100.00%)	1946/1946(100.00%)	3/15(20.00%)	14/897(1.56%)	1/5(20.00%)	10/653(1.53%)	**0/11(0.00%)**	**0/21270(0.00%)**	1/8(12.50%)	10/280(3.57%)

```java
1 Archive archive = new Archive(inputStream);
2 while (true) {
3     FileHeader fileHeader = archive.nextFileHeader();
4     if (fileHeader == null)
5         break;
6     /* infinite loop in extractFile */
7     archive.extractFile(fileHeader,
    ↪  OutputStream.nullOutputStream());
8 }
```

Figure 11: Case Study of CVE-2022-23596

Table 10: Statistics of Reported Bugs. Bugs are deduplicated manually.

	CVE	Uncaught Exception	Stack Overflow	Out of Memory	Infinite Loop
Apache PdfBox	Under Review	59	8	-	-
Apache POI	-	6	-	-	-
fastjson	-	15	7	-	-
iText 7	CVE-2022-24196/7	27	27	2	-
jackson	-	2	-	-	-
json-java	-	1	-	-	-
junrar	CVE-2022-23596	-	-	-	1
metadata-extractor	CVE-2022-24613	30	-	3	-
zip4j	-	11	-	-	-
SUM	4	151	42	5	1

automata-guided fuzz driver to solve this performance bottleneck. As shown in Fig. 10, the amount of compilation is reduced to one. By building a general model interpreter, the validating event sequence can be fed in as data. Note that the sequence can be represented as a naive DFA without branches and loops. Then the interpreter translates the sequence into real actions, e.g., calls an API, updates a variable, etc. For a given alphabet, only one fuzz driver needs to be generated and compiled. In practice, RUBICK can test thousands to tens of thousands event sequences per second, which is thousands of times faster than the initial implementation.

D Statistics of All Projects

Tbl. 7 lists the number of APIs of these tested attack surfaces. RUBICK evaluates the top six. `tika-jpeg` and `metadata-jpeg` represent the attack surface accepting JPEG input format in Apache Tika and metadata-extractor.

E List of the P-values in Evaluation

Tbl. 8 lists all p-values used in evaluation. The p-values are acquired from Mann-Whitney U-test [54]. The p-values which are smaller than 5e-2 are highlighted in bold. "-" represents that the p-values cannot be calculated since the data in both sides are equal, which also means one side is not statistically significant than the other side.

F Case Study of Control-Flow-Sensitivity

Fig. 11 shows a case which demonstrates how control flow sensitivity improves fuzzing performance. The case comes from one of our found bugs. It shows a fuzz driver accepting an input file which is of a certain archive format. Since the input file can archive multiple files, it iterates the file headers (`FileHeader`) to extract all contained files. The function `extractFile` (line 7) contains an infinite loop bug which can be triggered by malformed file headers. The loop iteration of `fileHeader` (line 2 – 7) increases the possibility to find this bug since the iteration can force the fuzz driver to apply `extractFile` on every file header. In other words, by adding execution states to differentiate the API invoking contexts, the iteration loop maximizes the exploitation of mutated input. If the fuzz driver is not aware of the states for loop, the mutations of the content which belong to non-first file headers will be wasted. *Conclusively, control flow sensitivity improves the performance of the fuzz drivers by providing additional states to distinguish different API invoking contexts.*

G Statistics of False Positive Rate

Tbl. 9 lists the statistics of false positive rate. The results support the conclusion discussed in Section 4.5. In most cases (5/6), RUBICK has the lowest false positive rate (0.00 %).

H List of Found Bugs

Tbl. 10 lists the bugs we found and reported for the fuzzed attack surfaces using the fuzz drivers generated by RUBICK. The listed bugs are already deduplicated. Till now, most bugs are fixed or under the fix plan of the vendors.