

Automated Finite State Machine Extraction

Abstract

Finite state machine (FSM) is a type of computation models widely used in various software programs. Extracting implemented FSMs has many important applications in network, software engineering and security. In this paper, we first study how FSMs are implemented in real-world software. We then design a static analysis tool, FSMExtractor, to identify and synthesize implemented FSMs. Evaluation using 160 software programs shows that FSMExtractor can extract FSMs with good coverage and accuracy.

1. Introduction

A finite state machine (FSM) is a mathematical model of computation, which performs a series of predetermined actions in reaction to the model inputs [35]. FSMs provide a concise and expressive way to describe program logic, so that they widely exist in various software programs, including network protocols, compiler and event-driven programs.

Automatically extracting implemented FSMs in a program has many important applications. First, the implementation of a FSM may be inaccurate. By comparing the extracted version with its original design, one can detect potential mistakes residing in an implementation [23]. Second, the network verification process depends on underlying FSM models in different components to validate the whole network properties (i.e. isolation, reachability). Generally, the FSMs fed into verification are largely handcrafted through manual inspection [10, 31], which is time-consuming and error-prone. Third, extracted FSMs can help developers and automated program analysis tools better understand program semantics, facilitating the building of future code debloating [14, 27, 28] and fuzzing techniques [5, 32, 42].

Unfortunately, there has not yet been an existing algorithm that can extract all implemented FSMs in a program. Static techniques [17, 38] can extract certain models implemented in a program, but their models are less concise and expressive than FSM. Dynamic techniques [1, 7, 22] view the whole programs as a blackbox and model it as one single FSM in a coarse granularity, failing to extract all FSMs and localize program code pertaining to an implemented FSM. Dynamic techniques highly depend on used inputs during FSM inference, lacking soundness and completeness.

In this work, we presents a tool FSMExtractor that can effectively extract FSMs in a program with good coverage and accuracy. Technically, FSMExtractor utilizes static analysis techniques, takes program code as input, and outputs a five-element tuple $(Q, \Sigma, \delta, s_0, F)$ describing each identified FSM.

We build FSMExtractor in two steps. First, we conduct an empirical study on how FSMs are implemented in the real world. After examining 25 FSMs in the CGC dataset [8],

we find that there are clearly code patterns for FSM implementations. For example, all of our studied FSMs are implemented in loops which do not take a constant trip count, and a state transition operation is control dependent on the current state. Second, we design static analysis routines for the code patterns. Our static analysis techniques can recognize suspicious FSM loops, recognize variables representing FSM states, and synthesize the five-element tuples. Our evaluation using 160 programs from three sources shows that FSMExtractor can identify all implemented FSMs with very few false positives.

2. Empirical Study

In this section, we first review the mathematical definition of FSM. Then, we describe our empirical study on how FSMs are implemented in real-world software.

Background. A finite state machine (FSM) is a mathematical computation model, which consists of several internal states and takes external inputs. At any time, a FSM can be only in one state. When a certain condition is satisfied, a FSM transits from one state to another. A FSM can be specified using a five-element tuple $(Q, \Sigma, \delta, s_0, F)$, where Q is a set of internal states, Σ is an input alphabet, δ is a set of transition functions, s_0 is the initial state, and F is a set of final states.

Real-World Implementation. We leverage the DARPA CGC dataset [8] to understand how FSMs are implemented in the real world. We choose the CGC dataset because it contains a large number of diverse programs simplified from real-world software and it is also widely used in security community [29, 30, 43].

To conduct the study, we first randomly sample 40 programs from the CGC dataset. Then, we manually inspect the sampled programs and deeply examine the implementation of FSMs. In total, we identify 25 implemented FSMs and treat them as the targets of our study. Figure 1 shows one such example. Function `cgc_parse_set()` takes string `right` as input and returns `true` if `right` matches regular expression `"|("[^"]*)"?)|"`. Figure 2 shows the underlying FSM. In total, the FSM contains six different states and nine possible state transitions.

To guide the implementation of FSMExtractor, our empirical study is mainly conducted from the following aspects.

First, what code constructs are used to implement the studied FSMs? Since our goal is to statically identify and extract implemented FSMs, we must know what code constructs to inspect. Not surprisingly, all our studied FSMs are implemented using a loop, like the `while` loop at line 13 in Figure 1. In each loop iteration, an implemented FSM processes an input and determines whether to stay in the current state or transit to a new state. The underlying intuition is that

```

1  typedef enum setState {
2      start = 0,
3      open_set,
4      close_set,
5      open_double,
6      close_double,
7      error
8  } setState;
9
10 bool cgc_parse_set(char * right) {
11     setState state = start;
12
13     while (*right && state != close_set) {
14         if (*right == '|') {
15             switch(state) {
16                 case start:
17                     state = open_set; break;
18                 case open_double: break;
19                 default:
20                     state = close_set; break;
21             }
22         } else if (*right == '"') {
23             switch(state) {
24                 case open_double:
25                     state = close_double; break;
26                 case open_set:
27                     state = open_double; break;
28                 default:
29                     state = error; goto end;
30             }
31         } else {
32             switch(state) {
33                 case open_double: break;
34                 default:
35                     state = error; goto end;
36             }
37         }
38         right++;
39     }
40 end:
41     if ( state != close_set ) {
42         return false;
43     }
44     return true;
45 }

```

Figure 1: A simplified FSM implementation from the CGC dataset.

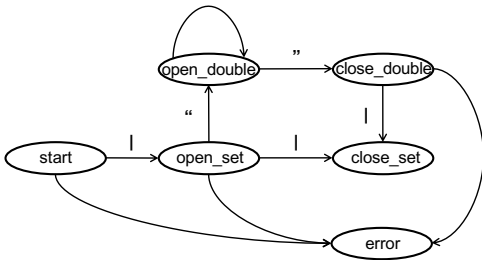


Figure 2: An implemented FSM in the CGC dataset.

a FSM usually needs to process multiple inputs and similar logics are applied during the processing, so that using a loop is a natural way to implement a FSM.

Another important observation is that the FSM loops do not execute constant iterations or take constant trip counts. Their executions dynamically depend on inputs, since it is very rare that a FSM can arrive at a final state after processing a predefined, constant number of inputs. For example, the iteration number of the loop in Figure 1 is not constant and it depends on the content of input string `right`.

Second, how internal states (Q) are maintained by the FSMs? Intuitively, there must be a state variable, which tracks the current state of a FSM. When state transition happens, the value of the state variable is changed. Our study confirms this intuition. We also find that state variables are either in integer type or enumeration type, and their values are discrete and bounded in a certain range. This finding indicates that static value set analysis [3, 12] can potentially determine all possible states of an implemented FSM. For example, local variable `state` declared at line 11 is the state variable of the FSM in Figure 1. It is in enumeration type. In total, it has six possible values specified by its type declaration at line 1, corresponding to the six states in Figure 2. Interestingly, one studied FSM loop contains two state variables, and this case reminds us that developers could use one loop to implement multiple FSMs. We need to extract all of them when designing FSMExtractor.

Third, what is the input alphabet (Σ)? The input alphabet of a FSM is theoretically bounded by all possible values of the data type used to represent inputs. For example, the input alphabet of the FSM in Figure 1 contains all possible byte values. There are also cases where an input alphabet is a subset of all possible values, and we think value set analysis can help refine the input alphabet for an identified FSM.

We observe that a FSM loop processes a distinct input in each iteration. Sometimes, a FSM loop needs to refer to a different memory location for a new input. Sometimes, a new input is written to the same location in each iteration before the FSM loop starts its procession. For example, `right` points to the input character processed by the FSM in each iteration. The value of `right` is incremented by one at line 38 in each iteration, so that the FSM reads a different memory location in each iteration.

We also observe that during the implementation of a FSM, developers usually do not enumerate the processing rule for every possible input value, and they tend to explicitly specify the rules only for several special values and leave others to be handled by a default rule. For example, only the processing rules for `'|'` and `'"` are explicitly specified in Figure 1, and all other byte values are handled by the default rule at line 31.

Four, how the transition functions (δ) are implemented? Transition functions take the current state and a value in the alphabet as input and output the next state. A transition function is executed in each iteration, the FSM relies on the output value to decide whether to transit to a new state. We observe that transition functions are implemented using control-flow constructs (e.g., `if`, `switch`). For example, a transition function in Figure 1 is implemented in line 14, 15, 16, and 17. If the current state is `start` at line 16 and the current input value is `'|'` at line 14, the transition function outputs `open_set` as the next state at line 17. Line 22, 23, 24, and 25 implement another transition function, which consumes an input character `'"` at line 22 and transits from

the current state `open_double` at line 24 to the next state `close_double` at line 25.

Five, how to specify the initial state (s_0) and the final states (F)? s_0 of a FSM can be specified by the value of the state variable before the execution of the FSM loop. For example, s_0 of the FSM in Figure 1 is `start`, which is the value of `state` before the loop execution at line 13. When a FSM loop finishes its execution, all possible values of the state variable represent F of the FSM. For example, the `while` loop in Figure 1 terminates its execution when finishing parsing string `right`, so that `state` can be any of the six values in the type declaration at line 1 and any state of the FSM in Figure 2 can be a final state.

To sum up, our empirical study shows that there are clearly code patterns used by developers to implement FSMs. In Section 3, we will discuss how we leverage these patterns to build FSMExtractor, which can automatically extract implemented FSMs from a program.

3. FSMExtractor Design

Our empirical study in Section 2 shows that a FSM is usually implemented in a loop which does not take a constant trip count (or iteration number) and conditionally updates a state variable to transit to a new state in each iteration. Therefore, FSMExtractor searches FSM loops by first filtering loops with constant trip counts (Section 3.1) and then identifying loops with state variable updates (Section 3.2). The ultimate goal of FSMExtractor is to construct implemented FSMs in a program, and thus we will discuss how FSMExtractor figures out the five-element tuple $(Q, \sum, \delta, s_0, F)$ for an identified FSM in Section 3.3.

Algorithm 1 shows the workflow of FSMExtractor. FSMExtractor takes the source code of a program as input and outputs the five-element tuple (line 8) and source code information (line 9) for each implemented FSM in the program.

Algorithm 1 Finite State Machine Extraction

Require: source code of a program: P

```

1: function FSMEXTRACTOR( $P$ )
2:   initialize an empty FSM set  $S = \{\}$ 
3:   for each loop  $l$  in  $P$  do
4:     if  $l$  takes a constant trip count then
5:       continue
6:     end if
7:     if  $l$  has state variable updates then
8:        $t(Q, \sum, \delta, s_0, F) \leftarrow \text{ConstructFSM}(l)$ 
9:        $\text{srcInfo} \leftarrow \text{ExtractSRCInfo}(l)$ 
10:       $S.\text{Insert}(t, \text{srcInfo})$ 
11:    end if
12:  end for
13:  return  $S$ 
14: end function

```

3.1 Filtering Loops with Constant Trip Counts

As discussed in Section 2, a FSM is usually implemented using a loop and the loop processes one input in each iteration to decide whether to transit to a new state. In reality, it is very rare that a FSM can arrive at a final state after processing a predefined, constant number of inputs. Our empirical study confirms this intuition. None of our studied FMS loops take a constant trip count. To sum up, given a loop which iterates a constant number in each execution, the loop is unlikely to be a FSM implementation.

We mainly leverage scalar evolution analysis [2, 20, 44] to identify loops whose trip counts are constant. Scalar evolution analysis can identify reduction variables inside a loop. Reduction variables are integer variables, whose values are updated with a constant delta in each loop iteration. For example, variable `right` is the only reduction variable inside the loop in Figure 1, since its value is incremented by one in every loop iteration. When a loop finishes its execution, the value change of a reduction variable is a multiplication of the iterations executed by the loop.

After identifying reduction variables inside a loop, FSMExtractor examines each exit condition of the loop and checks whether any of them is to compare a reduction variable with a constant number. If so, then the loops trip count is constant and FSMExtractor filters out the loop. For example, only the dereferenced value of the reduction variable `right` is used in exit conditions of the loop in Figure 1. Since `right` is the only one reduction variable inside the loop and it is not compared with a constant number in exit conditions, FSMExtractor does not filter out the loop and considers it as a potential FSM loop for further analysis.

3.2 Pinpointing State Variables

Our empirical study shows that state variables are either in integer or enumeration type and a FSM loop conditionally conducts a state transition in each iteration. Therefore, a FSM loop must contain at least one memory write to an integer (or an enumeration) variable. Since transition functions need to refer to the current state, a value assigned to a state variable in one iteration of a FSM loop needs to propagate to future iterations. Given a candidate FSM loop, FSMExtractor leverages live variable analysis [36] to identify possible state variables, which are integer variables updated inside the loop and have updated values live outside the loop or in the next iteration.

We illustrate this approach by taking the FSM in Figure 1 for example. Variable `state` is an enumeration variable and it is assigned with a new value at line 17, 20, 25, 27, 29, and 35 inside the `while` loop. The updated values are possibly read at line 15, 23 and 32 in the next iteration of the loop or at line 41 outside the loop, so that these values are live in the next iteration and outside the loop. Therefore, FSMExtractor considers `state` as a possible state variable.

We further eliminate false positives when identifying state variables by considering how a FSM conducts state transitions. As discussed in Section 2, a transition function refers to the current state to determine the next state. Therefore, defining the next state through writing a new value to a state variable is control dependent [11] on a predicate evaluation using the current value of the state variable. Take the FSM in Figure 1 as an example, transiting to state `open_set` at line 17 by assigning `open_set` to `state` is control dependent on the evaluation of “`state==start`” where the current value of `state` is read at line 15 and `start` is a constant. Transiting to `close_double` at line 25 is control dependent on the evaluation of “`state==open_double`”, and the current value of `state` is read at line 23.

FSMExtractor implements this mechanism through the following two steps. First, for each memory write to an integer (or enumeration) variable inside a candidate loop, FSMExtractor searches conditional branches inside the loop which the memory write is control dependent on. Second, FSMExtractor checks whether the condition of a searched branch is data dependent on the value of the same integer variable. For example, the memory write at line 17 is conducted on an enumeration variable and it is control dependent on the underlying conditional branch instruction for the `switch` at line 15 and the `case` at line 16. The condition of the branch is “`state==start`” and it is data dependent on the value of the same enumeration variable `state`. Therefore, FSMExtractor identifies `state` as a state variable.

3.3 Constructing FSMs

We consider a loop as a FSM loop, if it does not take a constant trip count and contains updates to a state variable. As discussed in Section 2, a FSM loop may contain more than one state variable. In this case, the loop is used to implement multiple FSMs. With a FSM loop and identified state variables inside the loop, FSMExtractor constructs a FSM for each state variable, by figuring out the five-element tuple $(Q, \Sigma, \delta, s_0, F)$.

To figure out all possible states (Q) of a FSM is equivalent to determine all possible values of its state variable. If a state variable is in enumeration type, FSMExtractor recognizes all its possible values by examining the declaration of the enumeration type. For example, FSMExtractor identifies all the six possible values of the state variable `state` (Figure 1) by inspecting the type declaration at line 1. If a state variable is an integer variable, FSMExtractor regards a constant value assigned to the state variable or compared with the state variable as a possible state. The current version of FSMExtractor only examines the function containing the analyzing FSM loop, so that it may miss some states. Future work could inspect the whole program by applying an interprocedural value set analysis to identify more states.

One iteration of a FSM loop processes a distinct input, such as a new character from a string or a new incoming package. For the new input, a FSM loop either refers to a dif-

ferent memory location or refers to the same location whose content is changed in each iteration. Take the FSM loop in Figure 1 as an example, `right` is a pointer pointing to input characters. `right` is incremented by one in each iteration at line 38, so that the FSM loop refers to a different memory location for an input to be processed in each iteration. After figuring out where a FSM locates its inputs, FSMExtractor understands the type of the inputs and considers all possible values in that type as the input alphabet Σ . For example, FSMExtractor recognizes the Σ as all possible byte values for the FSM in Figure 1.

FSMExtractor mainly relies on symbolic execution [4, 6] to synthesize transition functions (δ). Given a state variable, FSMExtractor conducts reachability analysis on CFG to search paths starting from an assignment site of the variable and ending at an assignment site. For each path, FSMExtractor applies symbolic execution to collect path constraints and utilizes a constraint solver to detect whether there are conflicting constraints among the path constraints and whether the path constraints conflict with the pre-condition that the state variable is equal to the assigned value at the starting site. If not, FSMExtractor successfully identifies a transition function, which transits a FSM from a state to another state pertaining to the values used at the starting assignment site and ending site respectively. FSMExtractor figures out the input value the identified transition function refers to by analyzing the collected path constraints.

We illustrate how FSMExtractor synthesizes transition functions using the FSM in Figure 1 as an example. Line $17 \rightarrow 13 \rightarrow 22 \rightarrow 23 \rightarrow 26 \rightarrow 27$ is a path from one assignment site of the state variable `state` to another assignment site. The path constraints are “`*right != NULL && state != close_set && *right == '' && state == open_set`”, which do not contain conflicting constraints. The path constraints do not conflict with the pre-condition “`state==open_set`” specified at the starting assignment site at line 17. Therefore, FSMExtractor identifies a transition function which transits the FSM from `open_set` to `open_double`. FSMExtractor figures out the input value used by the transition function as “`''`”, indicated by the path constraint “`*right == ''`”. Line $17 \rightarrow 13 \rightarrow 14 \rightarrow 15 \rightarrow 16 \rightarrow 17$ is another path identified by the reachability analysis. However, the path constraints (“`*right != NULL && state != close_set && *right == '|' && state == start`”) conflict with the pre-condition (“`state==open_set`”), and thus FSMExtractor does not consider this path indicates a transition function.

FSMExtractor computes s_0 and F through analyzing the value of the state value before the execution of the FSM loop and after the loop execution respectively. For example, the value of `state` is `start` before the loop in Figure 1 executes, so that s_0 of the FSM is `start`. The loop terminates when finishing parsing the input string pointed by `right`,

leaving state to be any value declared at line 1, and thus F consists of all states.

4. Experiment

In this section, we will describe how we set up experiments to evaluate FSMExtractor (Section 4.1) and present the experimental results (Section 4.2).

4.1 Methodology

Implementation and Platform. We implement FSMExtractor using LLVM-7.0.0 [19], and conduct our experiments on a Linux machine, with E5-2630 CPU, 32GB memory and 3.10 kernel.

Benchmarks. FSMExtractor is a tool to automatically extract FSMs implemented in a program. Since we build FSMExtractor using LLVM, our current implementation can only work on C/C++ programs. However, we believe that our algorithm is general enough to be extended to other programming languages.

To evaluate FSMExtractor, we collect C/C++ programs from three sources. First, we evaluate FSMExtractor on two programs collected in a CTF contest [34], one contains a FSM, and the other one does not. Second, we leverage the DARPA CGC dataset [8]. In total, there are 197 programs in the CGC dataset. As discussed in Section 2, we have already used 40 of them to conduct our empirical study, so that we use the remaining 157 programs in our evaluation. Third, we apply FSMExtractor to OpenVPN [37], which provides an implementation of virtual private network and is included in software packages of every released Linux version.

Source	# programs	avg. KLOC	# loops	# FSM loops
CTF	2	0.3	19	1
CGC	157	7.1	6607	59
OpenVPN	1	120	512	6

Table 1: Benchmark Information.

The benchmark information is shown in Table 1. In total, we use 160 different programs to evaluate FSMExtractor. Our benchmark set is a representative sample of real-world software, since each program is either a widely-used real application or a simplified program from real software. Our benchmark programs are diverse. They cover programs in small, medium and large sizes, with lines of code ranging from 0.3 thousand to more than 100 thousand. There are more than seven thousand loops inside our benchmark programs and accurately identifying FSMs among the loops is not easy. To sum up, we believe that our benchmarks are good enough to evaluate the effectiveness of FSMExtractor. **Evaluation Setting.** For all our benchmark programs, we manually examine all their loops and identify all FSM loops. As shown in Table 1, there are in total 66 FSM loops. Four FSM loops contain two state variables, and all other FSM loops contain exact one state variable. Therefore, there are in total 70 FSMs implemented in all our benchmarks. We

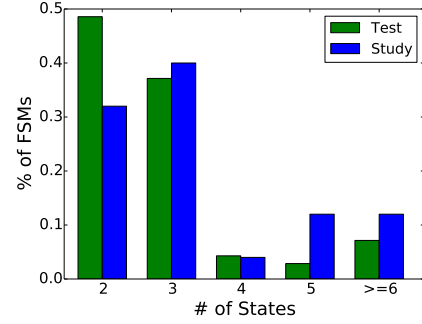


Figure 3: How the percentage of FSMs distributes across different state numbers.

apply FSMExtractor to all benchmark programs. We mainly compute metrics to answer two research questions regarding the coverage and accuracy of FSMExtractor.

Q1. Coverage: whether FSMExtractor identifies all FSMs?

Q2. Accuracy: whether FSMExtractor reports any false positive?

4.2 Experimental Results

Source	# FSM loops	# FSMs	# FNs	# FPs
CTF	1	1	0	0
CGC	59	63	0	2
OpenVPN	6	6	0	0

Table 2: Experimental Results.

Coverage. As shown in Table 2, FSMExtractor successfully identifies all the 66 FSM loops from the benchmarks. Since there are four FSM loops containing two state variables and FSMExtractor constructs a FSM for every identified state variable, there are in total 70 extracted FSMs. FSMExtractor has **no** false negative.

We then further inspect the characteristics of the identified FSMs. For most of the FSMs, their state variables are local variables. There are only three FSMs with a global variable as its state variable. Most FSMs use a standalone integer (or enumeration) variable as its state variable, and only three FSMs use a struct field as its state variable. This result shows that FSMExtractor can identify state variables implemented in various ways and developers tend to use a local, standalone, integer variable to represent the state of a FSM.

Figure 3 shows the percentage of the identified FSMs for different state numbers. More than 80% of FSMs contain either two states or three states. The percentage drops significantly when the state number is larger than three. There are two FSMs containing 11 states. These two FSMs have the largest state numbers among all the identified FSMs. On average, one identified FSM has 3.02 possible states. Figure 3 also shows how the percentage of FSMs distributes across different state numbers for the 25 studied FSMs. The average state number for the studied FSMs is 3.84. We use chi-square goodness of fit test to compare the two distribution

in Figure 3. The testing result shows that there is no significant difference between the two distributions under 99% confidence level. Our study results in Section 2 are general enough to be extended to other data sets.

Accuracy. As shown in Table 2, FSMExtractor is accurate. It only has two false positives. The false-positive-vs-FSM rate is 1:35. The two false positives are caused due to the same reason. In each case, an integer variable is used to hold a function pointer. The identified FSM loop checks whether the integer variable is 0. If so, the integer variable is assigned with a constant number, which is actually the address for the entry point of a function. FSMExtractor identifies the integer variable as a state variable. In the future, we plan to extend FSMExtractor by examining how identified state variables are used beyond FSM loops to filter out similar false positives.

5. Applications

In this section, we will discuss how the extracted FSMs can facilitate various network and security practices.

Network Verification. In network operation, before a network is deployed into production, its configuration needs to be verified to avoid runtime errors. In such a network verification process, network operators usually build behavior models for individual network appliances and then reason about the end-to-end properties of the network [10, 15, 16, 18, 21, 25]. FSM is an expressive behavior model to represent a wide range of network appliances, including switches and software network functions (such as load balancers, firewalls, and NAT). With individual FSMs and the network topology ready, the network operator could verify whether the communication between end hosts satisfies properties such as reachability, isolation, and loop-freedom.

FSMExtractor is helpful in the procedure of “building behavior models for individual network appliances”, which currently is manually crafted by the network operators by reading the code or according to their empirical understanding. FSMExtractor can primarily automate the transformation from network software to FSMs. More importantly, it provides the confidence that the output FSM is logically equivalent to the original software.

Code Debloating. Code bloat refers to code in unnecessarily large size [33]. It widely exists in production-run software [26]. If untackled, bloated code can introduce more bugs and vulnerabilities, lowering the security [13], and conduct resultless or redundant computation, degrading the performance [9, 40, 41].

Many techniques have been proposed to address the code bloating problem. They either remove temporary object copies [9, 24, 39–41] or eliminate functions unreachable from `main` [14, 27, 28]. None of them try to change the underlying program models. With extracted FSMs from FSMExtractor, further code debloating can be performed through eliminating unnecessary program logics. For example, given the ex-

tracted FSM in Figure 2, developers may consider removing state `open_double` and state `close_double`. A tool can take the FSM from FSMExtractor as input and automatically remove code pertaining to the two states. The tool can test or validate the changed program by monitoring the control flow inside the FSM loop and the value of the state variable.

Fuzz Testing. Fuzzing is an automated testing technique, which executes a program using randomly mutated inputs with the goal to trigger unexpected program behaviors, such as crashes and assertion errors [5, 32, 42]. Fuzzers are usually evaluated by measuring code coverage. A better fuzzer can cover more lines of code or branches under a given time constraint. The state-of-the-art fuzzers are not good at processing FSMs in a program. Take the FSM in Figure 1 as an example, “|” is the only input which has two characters and can transit the program to state `close_set` at line 20. If a fuzzer completely relies on random mutation, the probability to generate “|” is very low ($1/(256*256)$). However, if the fuzzer is enhanced by the FSM in Figure 2, it is very easy for the fuzzer to figure out how to create various inputs to quickly cover all states and state transitions.

6. Related Works

Program analysis to extract models. A set of work also applies various program analysis techniques to extract certain models implemented in programs: for example, NFactor [38] uses symbolic execution and program slicing to extract match-action tables in NF programs; StateAlyzr [17] extracts state abstractions from implementations of stateful protocols. Different from existing techniques, FSMExtractor extracts FSMs, which are more concise and expressive.

Blackbox modeling. Another approach to get the FSM of a program is blackbox modeling. L* algorithm invented by Angluin [1] is the theoretical foundation. It executes a program with different inputs and synthesizes the FSM by observing outputs. L* algorithm is applied in various scenarios (e.g., NF modeling [22], protocol analysis [7]). Compared with FSMExtractor, blackbox approaches suffer from two limitations. First, they model the whole program as one single FSM in a coarse granularity, while it is possible that there are multiple FSMs in a program. Second, their completeness and soundness are limited, since it is difficult to enumerate all inputs for FSM inference.

7. Conclusions

Automatically extracting FSMs in a program is important and challenging. In this paper, we tackle this problem by building a tool, FSMExtractor, which relies on static analysis to identify and synthesize implemented FSMs. Our evaluation shows that FSMExtractor successfully identifies all FSMs and reports very few false positives. In the future, we plan to combine FSMExtractor with existing network verification or fuzzing techniques to build end-to-end network or security applications.

References

- [1] D. Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 1987.
- [2] O. Bachmann, P. Wang, and E. Zima. Chains of recurrences - a method to expedite the evaluation of closed-form functions. *International Symposium on Symbolic and Algebraic Computing*, 1995.
- [3] G. Balakrishnan and T. W. Reps. Analyzing memory accesses in x86 executables. In *Proceedings of the 13th International Conference on Compiler Construction (CC '2004)*, Berlin, Heidelberg, 2004.
- [4] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI '08)*, Berkeley, CA, USA, 2008.
- [5] P. Chen and H. Chen. Angora: Efficient fuzzing by principled search. *Proceedings of the 39th IEEE Symposiums on Security and Privacy (Oakland '18)*, 2018.
- [6] V. Chipounov, V. Kuznetsov, and G. Candea. S2e: A platform for in-vivo multi-path analysis of software systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*, New York, NY, USA, 2011.
- [7] C. Y. Cho, D. Babic, P. Poosankam, K. Z. Chen, E. X. Wu, and D. Song. Mace: Model-inference-assisted concolic exploration for protocol and vulnerability discovery. In *Proceedings of the 20th USENIX Conference on Security (Usenix Security '11)*, Berkeley, CA, USA, 2011.
- [8] DARPA. Cyber Grand Challenge. URL: <https://archive.darpa.mil/CyberGrandChallenge/>.
- [9] B. Dufour, B. G. Ryder, and G. Sevitsky. A scalable technique for characterizing the usage of temporaries in framework-intensive java applications. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '08)*, New York, NY, USA, 2008.
- [10] S. K. Fayaz, T. Yu, Y. Tobioka, S. Chaki, and V. Sekar. Buzz: Testing context-dependent policies in stateful networks. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI '16)*, Berkeley, CA, USA, 2016.
- [11] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 1987.
- [12] W. Guo, D. Mu, X. Xing, M. Du, and D. Song. DEEPVSA: Facilitating value-set analysis with deep learning for post-mortem program analysis. In *Proceedings of the 28th USENIX Security Symposium (Usenix Security '19)*, Santa Clara, CA, 2019.
- [13] D. K. Hong, Q. A. Chen, and Z. M. Mao. An initial investigation of protocol customization. In *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation (FEAST '17)*, New York, NY, USA, 2017.
- [14] Y. Jiang, D. Wu, and P. Liu. Jred: Program customization and bloatware mitigation based on static analysis. In *IEEE 40th Annual Computer Software and Applications Conference (COMPSAC '16)*, 2016.
- [15] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI '12)*, Berkeley, CA, USA, 2012.
- [16] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*, Berkeley, CA, USA, 2013.
- [17] J. Khalid, A. Gember-Jacobson, R. Michael, A. Abhashkumar, and A. Akella. Paving the way for nfV: Simplifying middle-box modifications using statealzyr. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI '16)*, Berkeley, CA, USA, 2016.
- [18] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: Verifying network-wide invariants in real time. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*, Berkeley, CA, USA, 2013.
- [19] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO '04)*, Palo Alto, California, 2004.
- [20] LLVM. LLVMs Analysis and Transform Passes. URL: <https://llvm.org/docs/Passes.html>.
- [21] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with anteater. In *Proceedings of the ACM SIGCOMM 2011 Conference (SIGCOMM '11)*, New York, NY, USA, 2011.
- [22] S.-J. Moon, J. Helt, Y. Yuan, Y. Bieri, S. Banerjee, V. Sekar, W. Wu, M. Yannakakis, and Y. Zhang. Alembic: automated model inference for stateful network functions. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19)*, Berkeley, CA, USA, 2019.
- [23] M. Musuvathi and D. R. Engler. Model checking large network protocol implementations. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation (NSDI '04)*, Berkeley, CA, USA, 2004.
- [24] K. Nguyen and G. Xu. Cachetor: Detecting cacheable data to remove bloat. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (FSE '13)*, New York, NY, USA, 2013.
- [25] A. Panda, O. Lahav, K. Argyraki, M. Sagiv, and S. Shenker. Verifying reachability in networks with mutable datapaths. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*, Berkeley, CA, USA, 2017.
- [26] A. Quach, R. Erinfolami, D. Demicco, and A. Prakash. A multi-os cross-layer study of bloating in user programs, kernel and managed execution environments. In *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation (FEAST '17)*, New York, NY, USA, 2017.

- [27] V. Rastogi, D. Davidson, L. De Carli, S. Jha, and P. McDaniel. Cimplifier: Automatically debloating containers. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (FSE '17)*, New York, NY, USA, 2017.
- [28] V. Rastogi, C. Niddodi, S. Mohan, and S. Jha. New directions for container debloating. In *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation (FEAST '17)*, New York, NY, USA, 2017.
- [29] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS '17)*, San Diego, CA, USA, 2017.
- [30] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS '16)*, San Diego, CA, USA, 2016.
- [31] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu. Symnet: Scalable symbolic execution for modern networks. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*, New York, NY, USA, 2016.
- [32] wikipedia. american fuzzy lop (fuzzer). URL: [https://en.wikipedia.org/wiki/American_fuzzy_lop_\(fuzzer\)](https://en.wikipedia.org/wiki/American_fuzzy_lop_(fuzzer)), .
- [33] wikipedia. Code_bloat. URL: https://en.wikipedia.org/wiki/Code_bloat, .
- [34] Wikipedia. Capture the flag. URL: https://en.wikipedia.org/wiki/Capture_the_flag.
- [35] wikipedia. Finite-state machine. URL: https://en.wikipedia.org/wiki/Finite-state_machine.
- [36] Wikipedia. Live variable analysis. URL: https://en.wikipedia.org/wiki/Live_variable_analysis, .
- [37] Wikipedia. OpenVPN. URL: <https://en.wikipedia.org/wiki/OpenVPN>, .
- [38] W. Wu, Y. Zhang, and S. Banerjee. Automatic synthesis of nf models by program analysis. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks (HotNets '16)*, New York, NY, USA, 2016.
- [39] G. Xu. Finding reusable data structures. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '12)*, New York, NY, USA, 2012.
- [40] G. Xu and A. Rountev. Detecting inefficiently-used containers to avoid bloat. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*, New York, NY, USA, 2010.
- [41] G. Xu, M. Arnold, N. Mitchell, A. Rountev, and G. Sevitsky. Go with the flow: Profiling copies to find runtime bloat. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*, New York, NY, USA, 2009.
- [42] W. You, X. Wang, S. Ma, J. Huang, X. Zhang, X. Wang, and B. Liang. Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery. In *Proceedings of the 40th IEEE Symposiums on Security and Privacy (Oakland '19)*, 2019.
- [43] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim. Qsym: A practical concolic execution engine tailored for hybrid fuzzing. In *Proceedings of the 27th USENIX Conference on Security Symposium (Usenix Security '18)*, Berkeley, CA, USA, 2018.
- [44] E. V. Zima. On computational properties of chains of recurrences. In *Proceedings of the 2001 International Symposium on Symbolic and Algebraic Computation (ISSAC '01)*, New York, NY, USA, 2001.