Automated Finite State Machine Extraction

Yongheng Chen*
changochen1@gmail.com
NJU, Pennsylvania State University

Linhai Song
songlh@ist.psu.edu
Pennsylvania State University

Xinyu Xing
xxing@ist.psu.edu
Pennsylvania State University

Fengyuan Xu
fengyuan.xu@nju.edu.cn
Nanjing University

Wenfei Wu
wenfeiwu@outlook.com
Tsinghua University

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 the networking, software engineering and security domains. In this paper, we first conduct an empirical study to understand how FSMs are implemented in real-world software. Under the guidance of our study results, we then design a static analysis tool, FSMExtractor, to automatically identify and synthesize implemented FSMs. Evaluation using 160 software programs from three sources shows that FSMExtractor can extract all implemented FSMs and report very few false positives.

CCS CONCEPTS

• **Software and its engineering → Automated static analysis**; **Software testing and debugging**; • **Networks** → *Protocol testing and verification.*

KEYWORDS

finite state machine, static analysis

ACM Reference Format:
Yongheng Chen, Linhai Song, Xinyu Xing, Fengyuan Xu, and Wenfei Wu. 2019. Automated Finite State Machine Extraction. In *3rd Workshop on Forming an Ecosystem Around Software Transformation (FEAST'19), November 15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3338502.3359760

1 INTRODUCTION

A finite state machine (FSM) is a mathematical computation model that performs a series of predetermined actions in reaction to the model inputs [24]. FSMs provide a concise and expressive way to describe program logic, so that they are widely adopted in different types of software, including network protocols, compiler and event-driven programs.

Automatically extracting implemented FSMs from a program has many important applications. First, the implementation of an FSM may be inaccurate. By comparing the extracted version with its original design, one can detect potential mistakes residing in an implementation [28]. Second, the network verification process depends on underlying FSM models in different components to validate the whole network's properties (e.g. isolation, reachability). Right now, the FSMs fed into verification are largely handcrafted through manual inspection [15, 37], 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 [19, 33, 34] and fuzzing techniques [2, 11, 42].

Unfortunately, there has not yet been an existing algorithm that can extract all implemented FSMs from a program. Existing static techniques [22, 38] can extract certain models implemented in a program, but their extracted models are less concise and expressive than FSM. Dynamic techniques [6, 13, 27] view the whole program 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 inputs used during FSM inference, lacking soundness and completeness.

In this work, we present a tool FSMExtractor that can effectively extract implemented FSMs in a program with good coverage and accuracy. FSMExtractor is built based on LLVM infrastructure, and it takes LLVM intermediate code emitted during compilation as input. FSMExtractor utilizes static analysis techniques to search FSM implementations inside an input program and outputs a five-element tuple (Q, $\sum$, δ, 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 [1], we find that FSM implementations rely on certain code patterns. 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, pinpoint variables representing FSM states, and synthesize the five-element tuple for each identified FSM. We use 160 programs from three sources to evaluate FSMExtractor. The evaluation results show that FSMExtractor can identify all implemented FSMs with very few false positives.

2 UNDERSTANDING REAL-WORLD FSM IMPLEMENTATIONS

In this section, we first provide some background for FSM and the methodology of our empirical study. We then present detailed empirical study results.

*The work was done when Yongheng Chen was a visiting student at Pennsylvania State University.

FEAST'19, November 15, 2019, London, United Kingdom
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6834-6/19/11…$15.00
https://doi.org/10.1145/3338502.3359760

2.1 Background and Methodology

A finite state machine (FSM) is a mathematical computation model that takes external inputs and transmits among a set of predefined internal states. At any time, an FSM can only be at one state. When a certain condition is satisfied, an FSM transits from one state to another. An FSM can be specified using a five-element tuple (Q, $\sum$, δ, s_0, F), where Q is a set of internal states, $\sum$ 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.

We inspect the DARPA CGC dataset [1] 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 [35, 36, 43].

```
 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: An FSM implementation from the CGC dataset. ***The code has been simplified for illustration purpose.***

To conduct the study, we first randomly sample 40 programs from the CGC dataset. We then manually inspect the sampled programs and look for FSM implementations. 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 the string `right` as input and returns `true` if `right` matches regular expression "`|("[^"]*")?|`". Figure 2 shows the underlying FSM. In total, the implemented FSM contains six different states and nine possible state transitions.

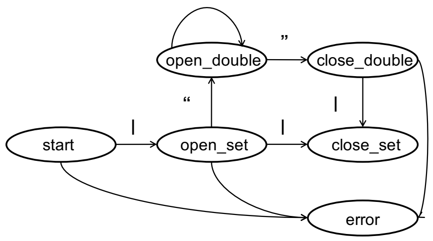

Figure 2: An implemented FSM in the CGC dataset.

2.2 Study Results

To guide the design of FSMExtractor, our empirical study is mainly conducted to answer the following five questions.

Q1. *what code constructs are used to implement the 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 an 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 an FSM.

Another important observation is that an FSM loop usually does not execute constant iterations or take a constant trip count. Its execution dynamically depends on its inputs. Intuitively, it is very rare that an FSM can arrive at a final state after processing a predefined, constant number of inputs. We take the FSM implemented in Figure 1 for example. The iteration number of the `while` loop is not constant and when the loop terminates its execution depends on the content of the string `right`.

Q2. *how internal states (Q) are maintained by the FSMs?* Intuitively, there must be a state variable to track the current state of an FSM. The value of the state variable changes when a state transition happens. 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 [8, 17] can potentially determine all possible states for an implemented FSM. For example, the local variable `state` declared at line 11 is the state variable of the FSM implementation in Figure 1. It is in enumeration type. In total, it has six possible values specified during its type declaration at line 1, corresponding to the six states in Figure 2. Interestingly, one studied FSM loop contains two state variables, which reminds us that developers could use one loop to implement multiple FSMs. We need to extract all of them when designing FSMExtractor.

Q3. *what are the input alphabets ($\sum$)?* The input alphabet of an FSM is theoretically bounded by 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 only a subset of all possible values in a particular type, and thus we think value set analysis should be applied to help refine the input alphabet of an identified FSM.

We have two observations about how an FSM loop handles inputs. First, an FSM loop processes a distinct input in each iteration. Sometimes, an FSM loop needs to refer to a different memory location for a new input. Sometimes, a new input is written to the same location before an FSM loop starts its procession in an iteration. For example, `right` in Figure 1 points to the input character processed by the FSM loop in each iteration. The value of `right` is incremented by one at line 38, so that the FSM loop reads an input from a different memory location in each iteration. Second, when implementing an FSM, developers usually do not explicitly implement the processing rule for every possible input value. Instead, they tend to specify rules only for several important input values and leave the others to be handled by a default rule. For example, only the processing rules for `'|'` and `'"'` are explicitly specified in Figure 1, and all the other input byte values are handled by the default rule at line 31.

Q4. *how the transition functions (δ) are implemented?* One transition function is executed in each iteration of an FSM loop. It produces the next state for the FSM to transmit to based on the current state and the current input value. 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 at 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.

Q5. *how to specify the initial states (s_0) and the final states (F)?* The initial state s_0 of an FSM can be specified by the value of the state variable before the execution of the corresponding FSM loop. For example, the initial state s_0 of the FSM in Figure 1 is `start`, which is the value of `state` before the loop execution at line 13. When an FSM loop finishes its execution, all possible values of its state variable constitute F. We take the FSM implemented in Figure 1 as an illustration. The `while` loop terminates its execution when finishing parsing the string `right`. At that time, `state` can be any of the six values specified during the type declaration at line 1. Therefore, any state of the FSM in Figure 2 can be a final state.

Discussion. Our empirical study shows that FSM implementations follow certain code patterns. For example, all our studied FSMs are implemented using a loop, and state transitions are implemented in a way that the next value of a state variable is control dependent on the current value of the state variable and the current input. This finding motivates us to leverage static analysis to identify FSM implementations through matching the code patterns. We will present more details later in Section 3.

Threats to the validity of our study could come from several aspects. All our studied FSM implementations come from the CGC dataset, which is designed for security purposes and only contains simplified programs instead of real software. Although all the studied FSMs are implemented in a similar way, we do believe that there are other methods to implement an FSM, such as using an event handler or a recursive function. Despite these limitations, all our findings are summarized after inspecting a relatively large number of randomly sampled FSM implementations. We believe our findings are intuitive and general enough to represent a high percentage of FSMs implemented in real-world software.

3 FSMEXTRACTOR DESIGN

Our empirical study in Section 2 shows that an 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 FSMs implemented in a program, and thus we will discuss how FSMExtractor figures out the five-element tuple (Q, $\sum$, δ, s_0, F) for each identified FSM in Section 3.3.

Algorithm 1 shows the workflow of FSMExtractor. FSMExtractor is built based on LLVM infrastructure, so that it takes LLVM intermediate code of a program as input. After analysis, FSMExtractor 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: LLVM IR 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($Q$, $\sum$, $\delta$, $s_0$, $F$) ← ConstructFSM($l$)
 9:             srcInfo ← ExtractSRCInfo($l$)
10:             S.Insert((t, 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, an 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 an 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 FSM 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 an FSM implementation.

We mainly leverage scalar evolution analysis [5, 7, 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 exit condition is to compare a reduction variable with a constant number. If so, then the loop's trip count is constant and FSMExtractor filters out the loop. We take the FSM implementation in Figure 1 as an illustration. Variable `right` is the only reduction variable inside the loop. None of the exit conditions of the loop compare `right` with a constant number, and only the value read from the memory location pointed by `right` is used in an exit condition. Therefore, 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 an FSM loop conditionally conducts a state transition in each iteration. Therefore, an 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 an FSM loop needs to propagate to future iterations. Given a candidate FSM loop, FSMExtractor leverages live variable analysis [9] to identify possible state variables, which are integer variables updated inside the loop and have updated values live outside the loop or in future loop iterations.

We illustrate this approach by taking the FSM implementation in Figure 1 for example. Variable `state` is an enumeration variable and it is updated with a new value at line 17, 20, 25, 27, 29, and 35 inside the `while` loop. These new 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 variable `state` as a potential state variable.

We further eliminate false positives when identifying state variables by considering how an 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 [16] on a predicate evaluation using the current value of the state variable. We 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 predicate evaluation of "`state==start`" at line 15, where the current value of `state` is read and `start` is a constant value declared at line 2. As another example, transiting to `close_double` at line 25 is control dependent on the evaluation of "`state==open_double`" at line 23, where the current value of the state variable `state` is read.

FSMExtractor implements this mechanism through the following two steps. First, for each memory write to an integer (or an 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 of the `switch` statement at line 15 and the `case` statement 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 an FSM loop, if it does not take a constant trip count and contains updates to a state variable. As discussed in Section 2, an FSM loop may contain more than one state variable. In this case, the loop is used to implement multiple FSMs. With an FSM loop and identified state variables inside the loop, FSMExtractor constructs an FSM for each state variable, by figuring out the five-element tuple $(Q, \sum, \delta, s_0, F)$.

To figure out all possible states (Q) of an 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` in 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 an analyzed 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 an FSM loop processes a distinct input, such as a new character from a string or a new incoming package. For the new input, an FSM loop either refers to a different memory location or refers to the same location whose content is updated before the FSM loop starts its processing. We take the FSM loop in Figure 1 as an example. Variable `right` is a pointer pointing to input characters. The value of `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 process in each iteration. After figuring out where an FSM locates its inputs, FSMExtractor understands the type of the inputs and considers all possible values in that type as the input alphabet $\sum$. For example, FSMExtractor recognizes $\sum$ as all possible byte values for the FSM implemented in Figure 1.

FSMExtractor mainly relies on symbolic execution [10, 12] 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 state variable and ending at an assignment site. For each path, FSMExtractor applies symbolic execution to collect path constraints and utilizes a constraint solver to validate the following two conditions. First, there are no conflicting constraints among the collected path constraints. Second, all collected path constraints do not conflict with the pre-condition that the state variable is equal to the assigned value at the starting assignment site. If the two conditions are satisfied, FSMExtractor successfully identifies a transition function, which transits the FSM from one state to another state pertaining to the values used at the starting assignment site and ending site respectively. By analyzing the collected path constraints, FSMExtractor can also figure out the input value processed by the identified transition function.

We illustrate how FSMExtractor synthesizes transition functions using the implemented FSM in Figure 1 as an example. Line 17 → 13 → 22 → 23 → 26 → 27 is a path from an 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 → 13 → 14 → 15 → 16 → 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`") specified at line 17, and thus FSMExtractor does not consider this path indicates a transition function.

FSMExtractor computes s_0 and F of an identified FSM through analyzing the value of the state variable before the execution of the corresponding FSM loop and after the execution of the loop respectively. For example, the value of `state` is `start` before the loop in Figure 1 executes at line 13, 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 of the FSM in Figure 2.

4 EXPERIMENT

In this section, we will describe our experimental settings in Section 4.1 and experimental results in Section 4.2.

4.1 Methodology

Implementation and Platform. We implement FSMExtractor using LLVM-7.0.0 [25], 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 in a CTF contest [3]. One program contains an FSM, and the other one does not contain any FSM. Second, we leverage the DARPA CGC dataset [1]. In total, there are 197 programs in the CGC dataset. As discussed in Section 2, we have already used 40 of them for our empirical study. Therefore, we use the remaining 157 programs in our evaluation. Third, we apply FSMExtractor to OpenVPN [4], which is an implementation of virtual private network. OpenVPN is widely-used and it is included in software packages of many 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.

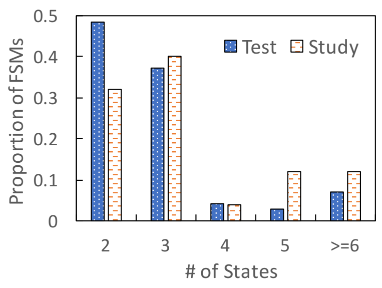

Figure 3: How the proportion of FSMs distributes across different numbers of states.

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. Accurately identifying FSM implementations 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 the other FSM loops contain only one state variable. Therefore, there are in total 70 FSMs implemented in all our benchmarks. We 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 an 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 identified FSMs. For most of them, their state variables are local variables. There are only three FSMs using global variables as their state variables. Most FSMs use standalone integer (or enumeration) variables to represent their states, and only three FSMs use `struct` fields as their state variables. These results show that FSMExtractor can identify state variables implemented in different ways.

Figure 3 shows how the proportion of FSMs distributes across different state numbers for the FSMs identified by FSMExtractor and the FSMs studied in Section 2. Most FSMs from the two sources contain two or three states, and the proportion drops significantly when the state number is larger than three. On average, one identified FSM has 3.02 states, and one studied FSM has 3.84 states. The average state numbers are similar, demonstrating that the sampled FSM set used in our study is general.

Accuracy. As shown in Table 2, FSMExtractor is accurate. It only has two false positives. The ratio of false positives over FSMs is 1/35. The two false positives are 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 zero. If so, the integer variable is assigned with a constant number, which is actually the entry address of a function. FSMExtractor mistakenly identifies the integer variable as a state variable and the loop as an FSM implementation.

4.3 Discussions and Limitations

All false positives reported by FSMExtractor are due to the fact that FSMExtractor does not consider how an identified state variable is used outside the corresponding FSM loop. We plan to extend FSMExtractor to eliminate similar false positives. The design of FSMExtractor is guided by the empirical study, and it is also limited by the study. One limitation is that the current version of FSMExtractor cannot detect FSMs implemented in a recursive function or an event handler. FSMExtractor is a static technique. It faces many traditional technical challenges (e.g. alias analysis, inter-procedural analysis). FSMExtractor may have both false positives and false negatives when analyzing pointer-heavy code. It can also miss FSMs whose state variable updates are conducted by an indirect callee deep in the call chain from the FSM loop. In the future, we plan to design practical heuristics or combine dynamic techniques with FSMExtractor to address these limitations.

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 [15, 20, 21, 23, 26, 30]. FSM is an expressive behavior model to represent a wide range of network appliances, including switches and software network functions (e.g. load balancers, firewalls, NAT). With individual FSMs and the network topology ready, the network operator could verify whether the communication between end hosts satisfies properties (e.g. reachability, isolation, 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 codes in an unnecessarily large size [31]. It widely exists in production-run software [32]. If untackled, bloated codes can introduce vulnerabilities [18] and degrade the software performance [14, 40, 41]. Many techniques are proposed to address the code bloating problem. They either remove temporary object copies [14, 29, 39–41] or eliminate functions unreached from `main` [19, 33, 34]. None of them 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 extracted FSM in Figure 2, developers may consider removing state `open_double` and `close_double`. A tool can take the FSM 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 in 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 [2, 11, 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. We take the FSM in Figure 1 as an example. String "`||`" 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 ($1/(256 \times 256)$). However, if the fuzzer is enhanced by the FSM in Figure 2, it will easily figure out how to create inputs to quickly cover all states and state transitions.

6 RELATED WORKS

Program analysis to extract models. A set of work also applies 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 [22] 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 [6] 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 [27], protocol analysis [13]). 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 implemented 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] Cyber Grand Challenge. URL: https://archive.darpa.mil/CyberGrandChallenge/.

[2] American fuzzy lop. URL: http://lcamtuf.coredump.cx/afl/.

[3] Capture the flag. URL: https://ctftime.org/ctf-wtf/.

[4] OpenVPN. URL: https://openvpn.net/.

[5] LLVM's Analysis and Transform Passes. URL: https://llvm.org/docs/Passes.html.

[6] D. Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 1987.

[7] 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.

[8] G. Balakrishnan and T. W. Reps. Analyzing memory accesses in x86 executables. In *Proceedings of the 13th International Conference on Compiler Construction (CC '04)*, Berlin, Heidelberg, 2004.

[9] M. Bozga, J.-C. Fernandez, and L. Ghirvu. State space reduction based on live variables analysis. In *Static Analysis (SAS '99')*, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[10] 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.

[11] P. Chen and H. Chen. Angora: Efficient fuzzing by principled search. *Proceedings of the 39th IEEE Symposiums on Security and Privacy (Oakland '18)*, 2018.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 1987.

[17] W. Guo, D. Mu, X. Xing, M. Du, and D. Song. DEEPVSA: Facilitating value-set analysis with deep learning for postmortem program analysis. In *Proceedings of the 28th USENIX Security Symposium (Usenix Security '19)*, Santa Clara, CA, 2019.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] J. Khalid, A. Gember-Jacobson, R. Michael, A. Abhashkumar, and A. Akella. Paving the way for nfv: Simplifying middlebox modifications using statealyzr. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI '16)*, Berkeley, CA, USA, 2016.

[23] 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.

[24] T. Koshy. *Discrete Mathematics With Applications*. Academic Press, 2004.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] R. Poli. A simple but theoretically-motivated method to control bloat in genetic programming. In *Genetic Programming (EuroGP '03)*, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[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.