FiniteFuzz : Finite State Machine Fuzzer For Industrial Control IoT Devices

Jaskaran Kaur

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science and Applications

Matthew Hicks, Chair

Ali R. Butt

Changwoo Min

May 2, 2023

Blacksburg, Virginia

Keywords: Fuzzing, Finite State Machine, Industrial Control Systems

FiniteFuzz : Finite State Machine Fuzzer For Industrial Control IoT Devices

Jaskaran Kaur

(ABSTRACT)

Automated software testing techniques have become increasingly popular in recent years, with fuzzing being one of the most prevalent approaches. However, fuzzing Finite State Machines (FSMs) poses a significant challenge due to state and input dependency, resulting in exponential exploration time required to unlock the Finite State Machine. To address this issue, we present a novel approach in this research paper by introducing FiniteFuzz, a Grey Box Fuzzer explicitly designed to fuzz Finite State Machines. Unlike the Blackbox fuzzers, FiniteFuzz employs a mutational technique that utilizes feedback to steer the fuzzing process. FiniteFuzz takes a random set of states and compares them with the desired FSM and records the states that increase the coverage of the Finite State Machine. The next seed incorporates the feedback received from all the previous seed inputs. This avoids exploring the same path multiple times and results in linear performance for all the types of Finite State machines possible. Our findings reveal that the use of FiniteFuzz significantly reduces the exploration time required to uncover each state of the machine, making it a promising solution for generating Finite State Machines. We tested our FiniteFuzz on 4 different types of Finite State Machines with each scenario resulting in at least 5X performance improvement in FSM generation. The potential applications of FSMs are vast, and our research suggests that the proposed approach can be used to generate any type of Finite State Machine.

FiniteFuzz : Finite State Machine Fuzzer For Industrial Control IoT Devices

Jaskaran Kaur

(GENERAL AUDIENCE ABSTRACT)

Fuzzing, also known as Fuzz testing is a technique used to test software for security vulnerabilities, errors, and unexpected behavior. It involves generating random or semi-random input to a software application such as an operating system, or network service to test how it responds. Once input is generated, it is sent to the target application, which may crash, hang or produce unexpected results in response to the input. The results are then analyzed to identify potential vulnerabilities such as buffer overflows, input validation errors, and resource leaks. Fuzzing is also used to test software that is difficult to test through other means, such as closed-source software or embedded systems. We generated a Fuzzer,FiniteFuzz for Finite State Machine that unlocks the FSM starting from the random input and exploring only those seeds that increases the test coverage.

Contents

List of Figures

Chapter 1

Introduction

1.1 Introduction

As technology continues to advance and integrate with various industries, the risks of software vulnerabilities have become more significant. Hackers are continually looking for new ways to exploit these vulnerabilities, which can pose a significant threat to our daily lives. However, these risks are not limited to software vulnerabilities alone. Industrial Control Systems (ICSs) [24]that rely on Programming Logic Devices [2] also face security risks that can have severe consequences.

Programming Logic Devices play a fundamental role in Industrial Control Systems (ICSs) by providing necessary automation and managing critical infrastructures. A decade back, these devices were not connected to the internet and operate in a separate network. But as information technology and industrialization [9, 20] continue to integrate, an increasing number of ICSs systems are being exposed to the internet.

ICs and PLCs communicate through network protocols [25, 27] which are typically implemented with minimal security and therefore vulnerable to internet attacks. Physical access to the machine is not even necessary for an attacker to exploit the vulnerabilities. **Heartbleed** is a great example of a security vulnerability that affected the widely used OpenSSL library used for secure communication over the Internet. The vulnerability existed for 2 years

in a library that is used by **the 66% of the Internet**. This incident highlighted the importance of in-depth testing of the Software which is not possible just by manual test cases. The end-to-end understanding of the protocols is difficult as the protocol documentation is a few pages long, any misinterpretation of this protocol can result in these vulnerabilities [22]. Also, at each state, the protocol is dependent on the current state which is dependent on earlier states, and the current input which means a huge set of inputs, which is difficult to achieve with the manual test cases. It is crucial to analyze the communication between ICs and PLCs, which means analyzing the protocols. These protocols are implemented using Finite State Machines [12]. To understand the vulnerabilities, we need to mimic the communication which is based on the Finite State Machines.

There has been a large body of research dedicated to FSM inference of network protocols, and some representative research from academia is available, which are listed as follows.

[19] analyzes the communication with Control Systems and IDE to identify security risks. After establishing a handshake between protocols and IDE [17], it splits the protocol and filters out the necessary information, and passes this to the Black box fuzzer to mimic the protocol, the unexpected changes are detected and monitored. However, this fuzzing tool does not mutate the input seeds that lead to unexpected behavior. Also, the author uses the Black box approach used to mutate the seeds and the time to reach the desired state.

[31] a fuzzer for OPC Protocol [14] that generates random input to the OPC server and monitors the response. A random input seed is generated taking care of the protocol border and the output is monitored using heartbeat, process state monitoring [11], and log recording, if any abnormal behavior is encountered, they capture the input. However, they get the abnormal behavior by randomly exploring the inputs, which would have taken a long time to get the desired state. The time to get this random behavior could have drastically improved if they use feedback with the Black box fuzzing [3].

[23] fuzzer test cases are generated keeping a balance of exploration and exploitation [10] monitoring the input seed's mutation. They prioritize the test cases that are more likely to cause state transitions in the fuzzing targets instead of random mutations. For seeds that explored the new paths, they increase the likelihood of choosing the same message for mutation in the future else lower the chances of this input getting selected. However, we believe the implementation of this algorithm is overly convoluted, they could have chosen a sequential order for generating the input as FSM follows that order and instead of assigning a higher weight to the input that explored a new path, the input should have been explored more.

[33] a smart fuzzing technology is proposed for Modbus-TCP [28], an adaptive algorithm for test case generation on the feedback from the target. A Smart Data Generator mutates the input seeds and anomalies are detected using Anomaly Monitoring [1] Module based on the return code and exception codes. The test cases are adjusted based on the return/exception code. However, they just mentioned they will use return code as feedback to generate the new test cases but they didn't clarify how will they use this.

To mimic the Finite State Machine, we use *fuzzing*. It is a revolutionary technique that has changed the way we approach software testing. It involves sending random or semi-random input to a program to identify vulnerabilities and programming errors. By generating unexpected inputs, fuzzing [13, 15] can expose flaws in software that traditional testing methods might miss [18].

As the implementation details of these protocols are very complex, black box fuzzing [32] is often considered to be the only practical approach. However, randomly generating inputs can be very inefficient as it can take a long time to reach a target state and we will be exploring one input multiple times.

This research explores a fresh algorithmic, FiniteFuzz approach that utilizes feedback to create thorough test cases for the Finite State Machines of Industrial Control Systems. Instead of doing random mutation, we find the test cases which increase the coverage and explore those test cases. The FiniteFuzz performs better compared to the traditional Black Box Fuzzer for all the FSMs.

In summary, this paper contributes the following:

- We examine the different possible types of Finite State Machines and implemented an algorithm to generate these Finite State Machines.
- We tested the Finite State Machine against the *Black-Box Fuzzer* which randomly generates the states of FSMs.
- We implemented a separate Grey Box fuzzer, FiniteFuzz for each category of FSM and created a test suite to compare it with the Black Box fuzzer
- Since inputs are random, we implemented the **Minimisation algorithms** for the input that ensure 100% coverage with minimum input seeds possible.

1.2 Background

This section is dedicated to presenting the core topics related to FiniteFuzz: *Coverage Guided Fuzzing*, and *Finite State Machines*.

1.2.1 Coverage-guided Fuzzing

Coverage Guided Fuzzing (CGF) is a type of software testing technique used to identify vulnerabilities and defects in a program. It involves generating and executing a large number of test cases, called "Fuzz Inputs" to find bugs or trigger crashes.

The *key concept* behind CGF [7, 8] is to use the feedback from the previous test cases to guide the generation of new inputs. In particular, CGF tools monitor the code coverage achieved by each test case and use this information to generate new test cases that explore the untested parts of the program. This approach is effective because it focuses testing efforts on the part that are most likely to contain bugs.

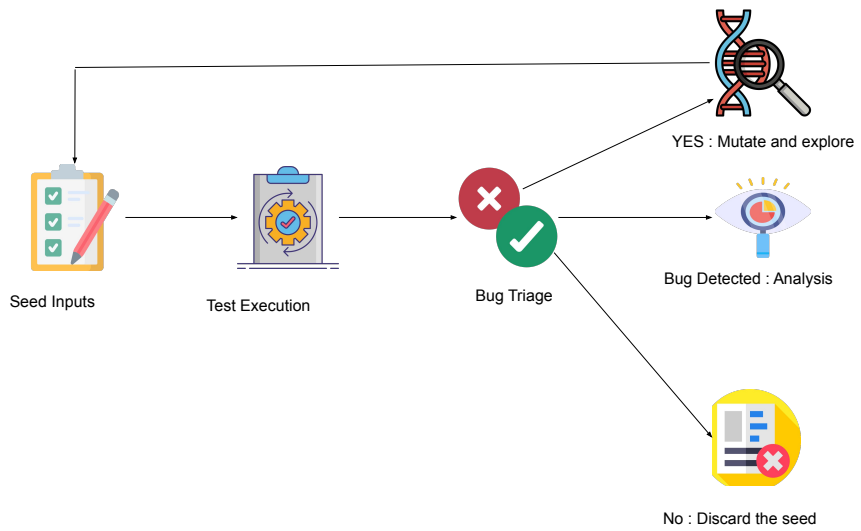


Figure 1.1: FiniteFuzz: Visualisation of Fuzzing Overview

CGF works in several stages -

1. **Input Generation:** The first step in CGF is to generate a set of initial inputs to use as test cases. These inputs can be randomly generated [32], pre-defined, or mutated from the existing inputs [16] The goal is to create a diverse set of inputs that cover different parts of the program.
2. **Test Execution:** The input is executed against the fuzzer. As the program runs, the coverage information is collected and recorded for each input. If a crash or other abnormal

behavior occurs, the tool can capture and report it.

3. **Bug Triage:** The tool then generates new test cases by mutating the existing inputs in the test suite. The tool prioritizes the new test cases based on their expected coverage. Test cases that are expected to cover new or untested parts of the program are given higher priority.

1.2.2 Finite State Machines

Finite State Machines(FSMs) [29] is a fundamental concept in Computer Science and is widely used in software Development for tasks such as parsing output and controlling program flow. An FSM is a mathematical model used to represent systems that can be in one of a finite number of states.

Applications of Finite State Machines: FSMs have been used extensively in Computer Science and engineering to model and design systems such as software programs, digital circuits, and communication protocols [6]. One of the practical applications of FSMs is in the field of software testing, where they can be used to generate test cases that effectively exercise the behavior of a software program. Fuzzing, which is a technique used to test software systems by generating inputs that are designed to trigger unexpected behavior, is one example of how FSMs can be used in software testing.

The use of FSMs in fuzzing has been the subject of much research in recent years. Techniques such as model-based fuzzing [4, 21], where a software design is tested against a model of its expected behavior as an FSM, have shown promise in improving the effectiveness and efficiency of Fuzz testing.

Moore and Mealy FSMs [5, 26] are two commonly used types of FSMs in fuzzing. Moore machines have outputs that depend only on the current state of the machine, while Mealy

machines have outputs that depend on both the current state and the input to the machine. These two types of machines can be used to represent different aspects of the software system being tested and can be used in different ways to generate effective test cases.

Despite the progress that has been made in using FSMs in fuzzing, there are still many open research questions related to generating Finite State Machines in an effective way using Fuzzing. This paper aims to address this issue by proposing an algorithm Finite Fuzz that generates Finite State Machines in a very efficient and effective way.

1.3 Motivation

Software testing is an essential aspect of software development, ensuring that applications are functional, secure, and reliable. One popular technique for software testing is *fuzzing*, which involves generating random inputs to identify vulnerabilities or unexpected behavior in a system. It has proven to be an effective method for identifying software defects and security flaws, which could otherwise go unnoticed in traditional testing methods.

In the IoT world, Industrial Control Systems(ICSs) and Programming Logic Devices communicate through protocols that are implemented with very less security and are vulnerable to internet attacks. So, it is important to analyze the communication between these devices which means analyzing the protocols.

All the previous studies try to analyze the communication between these devices by sending inputs and observing the response from these devices. There has been very little explanation of how they are generating these inputs and the existing studies use a Black Box Fuzzer for generating these inputs. Since there are a lot of states in the protocols of the FSMs, randomly generating these inputs takes a lot of time in unlocking the Finite-state Machine. But none

of the studies discuss the time to generate the input seed for the protocol and the time to unlock the protocol completely.

In this paper, we aim to contribute to this research by introducing a novel approach, **Finite-Fuzz** to fuzzing that instead of purely relying on random inputs, checks for the new coverage in the input seed and explore only those seeds that increase the coverage of the FSMs under test. This approach enables us to generate highly *effective input seeds* that can identify previously unknown vulnerabilities.

Chapter 2

Fuzzing Tool

2.1 Technique

We designed a new Fuzzing tool, FiniteFuzz to generate Finite State Machines(FSMs) for testing software systems. It is based on *Grey Box fuzzing*, if we detect new states with the seed input, it captures the new states and uses this as the feedback to generate the next states. We keep on exploring the new states that increase the FSM under test coverage until we have *100% coverage*. We used the number of inputs as the measure to compare FiniteFuzz with the Black Box Fuzzer.

2.1.1 Description

To *achieve the goal of generating FSMs* effectively and efficiently, our tool, FiniteFuzz has been designed with several key features. *One such feature* is its ability to generate all states of the FSM in as few inputs as possible. This is achieved through the use of feedback-based techniques that enable our tool to efficiently navigate state transitions, thereby reducing the number of inputs required to generate the FSM. *In addition*, our tool is capable of generating complex behavior, which enables us to test real-world Finite State Machines that have intricate transitions. This is achieved by considering the variable length of each state which is what we see in real life and it is important as many software systems have complex

logic and state transitions that can be difficult to test effectively.

Another key feature of the FiniteFuzz tool is its ability to identify code coverage gaps. This is achieved by studying different Finite State Machines and dividing them into different categories. A huge number of different test cases were generated for each type and captures the number of inputs to reach that state. A separate Fuzzer with the same feedback logic is developed for each type of Finite State Machine. By doing so, our tool can identify areas of the software system that have not been adequately tested and can be vulnerable, allowing developers to focus their efforts on improving the quality of those areas.

Overall, our tool FiniteFuzz represents a significant improvement over existing techniques for generating FSMs. By *incorporating feedback-based techniques and the ability to generate FSM with complex behavior,* our tool is capable of generating comprehensive test cases that improve the overall quality and security of software systems.

2.1.2 Input Format

To ensure our FSM-generating tool can handle FSMs of varying sizes and complexities, we designed our inputs with careful consideration. Each input is designed to represent different states with each having a fixed size. We used a *string binary format to represent each state*, with each bit corresponding to a specific state variable.

For example, if we are generating an FSM with 5 states and a size of 4 for each state variable, the input would look something like this: ”0101 1010 1100 0000 1000”. To generate the inputs, *we used the rand() function of Mathematics along with the Bit Manipulation operation.* This approach allowed us to randomly generate states that exhibit complex behavior, thereby enabling us to better simulate real-world scenarios.

It’s important to note that the input state varies depending on the type of FSM being

generated but the underlying idea remains the same. By designing our inputs with careful consideration and using a string binary format to represent each state, we were able to generate FSMs of varying sizes and complexities efficiently and effectively.

2.1.3 Fuzzing Strategy

FiniteFuzz uses a *mutation-based strategy* to generate test cases for the FSM. We fix a desired finite state machine, let's call output. The fuzzing process begins by selecting an initial random sequence for the state machine, let's say input. The input is compared with the output and good states are captured. These good states are mutated by randomly modifying the mismatched symbols. The needed seed will have feedback from all the previous states. We keep on generating the inputs till we have 100% coverage i.e. we explored all the states of the Finite State Machine.

2.1.4 Fuzzing Feedback

FiniteFuzz uses feedback from the FSM to guide the fuzzing process. Specifically, after each input sequence is executed by the FSM, FiniteFuzz checks whether the state machine has transitioned to a new state. If a new state has been reached, our tool records this information and uses it to guide the fuzzing process. *For example,* if the FSM transitions to a new state that has not been visited, it captures the state and adds it as a good state, and mutates the rest of the states.

Here are the steps -

1. **Get Matched states:** The seed FSM is compared with the Target FSM in sequential

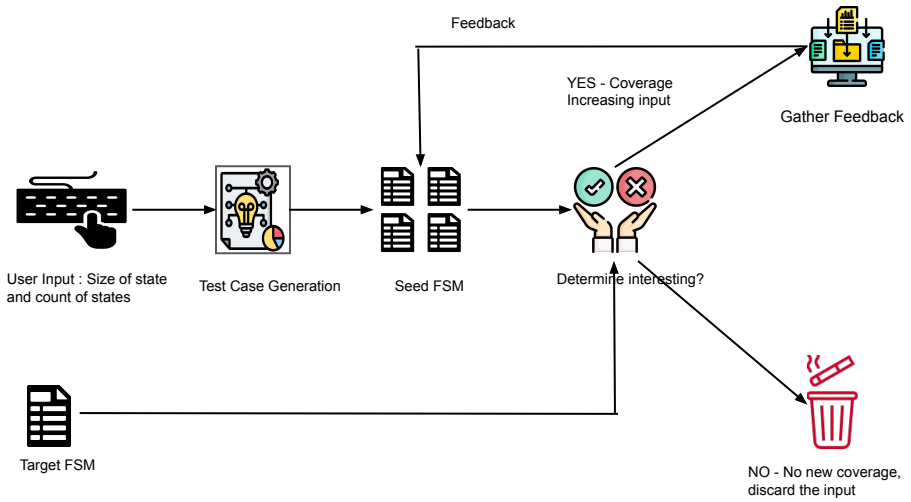


Figure 2.1: Visualization of Step-by-Step FiniteFuzz process.

order and the matched states and the count of the matched state are captured.

2. **Capturing the new feedback:** If there is new feedback, we modify the input seed with all the old feedback and the new feedback.
3. **Mutate the rest of the states:** The input seed has all the feedback received so far, for the remaining states, randomly replace the rest of the states with different symbols.

2.1.5 Coverage Analysis

FiniteFuzz keeps on executing till we achieved *100% test coverage i.e. we have covered all the states in the Finite State Machines.* Specifically, after the state is discovered by the FSM, it captures the state and the next seed will explore the new states of the Finite State Machine only. With the FiniteFuzz, we were able to achieve the full coverage in a linear time and it follows the same time irrespective of the number of states in the Finite State

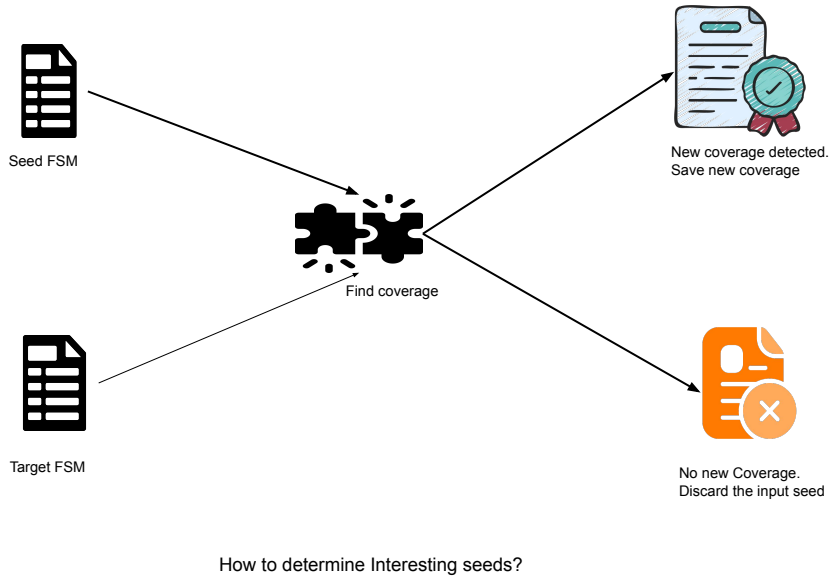


Figure 2.2: Visualization of how Fuzzer checks Interesting Seeds - Seeds that increases coverage.

Machines.

2.2 Implementation

In this section, we describe the approach used to carry out the *FSM Fuzzing experiments*-

2.2.1 Fuzzing Tool Selection

Since randomly generating input seeds and comparing them with the desired Finite State Machine takes exponential inputs, we need to think out of the box. We implemented **a Grey Box Fuzzer -FiniteFuzz that captures the feedback from the previous seeds**. It discards the input seed if there is no new coverage by this seed and only explores the

input seed if it results in increasing the coverage. In this way, we are not wasting any cycle in exploring a state multiple times and we are always moving in the forward direction. The Fuzzer stops when we have explored all the states of the Finite Machine. The random input generation was taking exponential input seeds with the Blackbox fuzzer but since the FiniteFuzz uses the feedback from all the previous input states, the expectation from the FiniteFuzz is to reduce these exponential inputs to linear inputs.

2.2.2 Test Suite Selection

We explored different types of Finite State Machines and *divided them into 6 categories.* For each type, we implemented an algorithm to generate the desired Finite State Machine from the random seeds. These categories are -

1. **Monotonic Finite State Machine(Only Go Forward)** - All the FSMs which only *go forward fall into this category.* In these FSMs, we move in sequential order, after discovering one state, we look for the next stage match, and so on.

   For example, Traffic Light Signal, "red", "yellow", and "green". After "yellow" only "green" and not "red". Another example is a Washing Machine which controls the washing and drying of clothes. It moves forward to the series of states - filling the water, washing the clothes, rinsing the clothes, and drying the clothes.

   For this type of FSM, we fix a desired FSM that we wanted to unlock, and for the input seed, we choose a random 64 states. If the input seed state matches the desired state, we move forward else we stay in the same state. If with this input seed we are able to unlock all the states of FSM, we stop fuzzing else we again start from random 64 symbols.

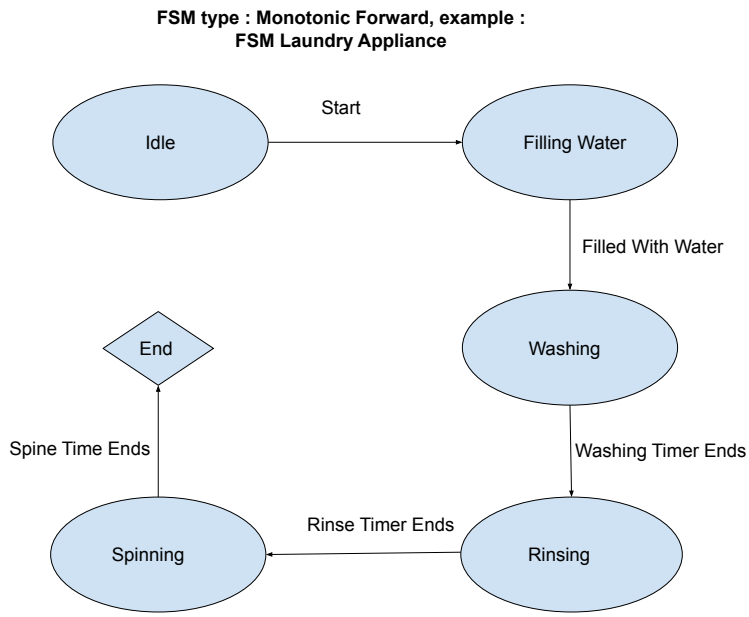


Figure 2.3: Example of Monotonic FSM

2. **Special Sequence with Reset on Miss** - These are *monotonic Finite state machines with a condition that the FSM gets reset if any mismatched symbol is encountered. For example,* Security Systems use an FSM to control access to a restricted area. The input sequence could be a presentation of a valid identification card and the input is entering a correct password and the output would be the opening of the door. If the sequence is not contiguous, the security system could reset to its original state. Another example is ATM which has states such as idle, input, processing, and output. When the ATM is idle, it is waiting for a user to insert a card and enter the pin. When the user adds the card and the PIN, the machine transition through the input state and verifies the information. If the information is invalid, it resets to the idle state

   For this type of FSM, we fix a desired FSM that we wanted to unlock, the input seed is of the same size as the FSM we wanted to achieve. If there is a mismatch between the input state and output FSM, we reset the sequence and generate a new random FSM to

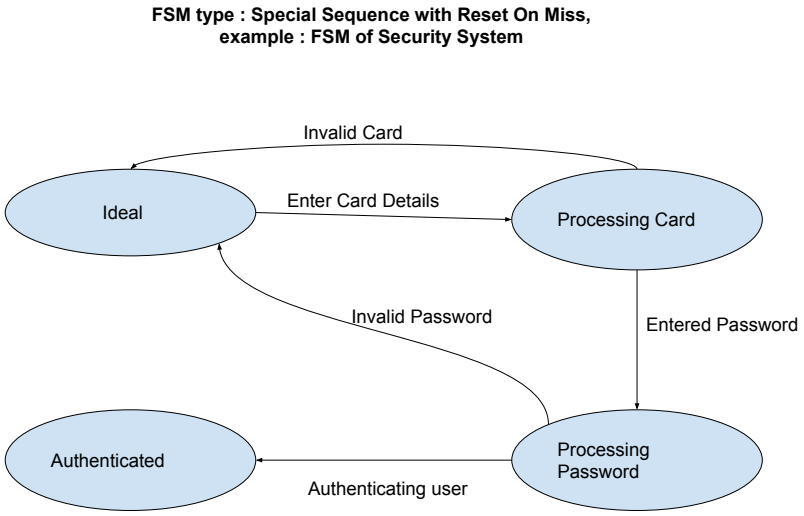


Figure 2.4: Example of Special Sequence with Reset on Miss FSM.

start with.

3. **Special Sequence with Reset after a fix misses** - These are similar to type 2 but *instead of resetting after the first mismatch, we have a threshold value,* if the mismatch exceeds the threshold, we reset the sequence. For this type of FSM, we fix a desired FSM that we wanted to unlock, the input seed size is the sum of the size of the output and the threshold. If the symbol matches, we move forward to the next state, if not we increase the mismatch symbols count, if it is greater than the threshold, we generate another input seed. *For example,* Speech Recognition software uses an FSM to process audio signals and recognizes spoken words. The FSM occasionally misinterprets the word or phrase. A threshold of error is allowed, if we bypass this threshold, some modification in the logic is added, and re-run the FSM.

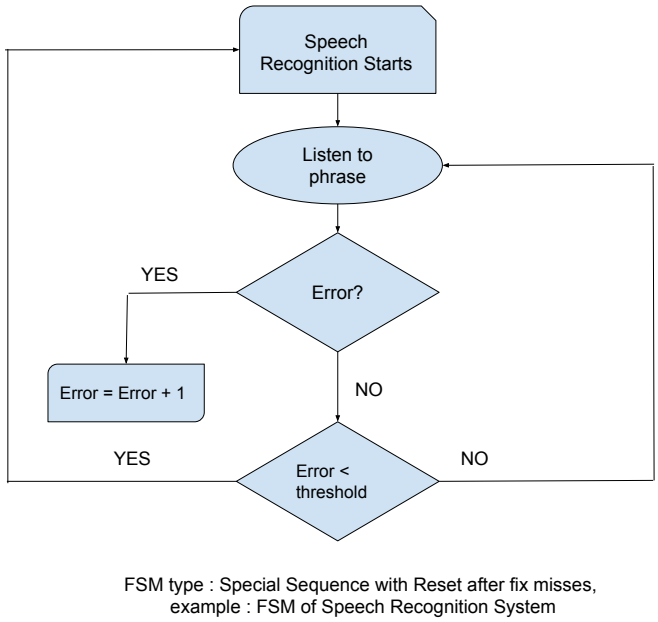


Figure 2.5: Example of Special Sequence with Reset after fix misses FSM.

4. **Looping Sequence with Special Exit Condition** - All the FSMs that keep on *moving in a loop until a special exit condition is encountered* fall under this category. Examples of these FSMs include -

   For example, Automatic Toll Booth has finite states such as idle, collecting the cash, and finished. When the toll is idle, it is waiting for the car to approach. When the car approach, the toll booth transitions to the collecting state, where it collects the appropriate toll from the driver. After that, it transitions to the finished star and shuts off. If another vehicle approaches, it follows the same process. The special exit condition could be when the machine is out of change or it runs out of paper. Another FSM is an Automatic Irrigation System that has a sequence of states that represents the process of monitoring and watering the plants. The special condition could be manually turning off the Irrigation System.

   For this FSM type, we set a random output Finite State Machine, and the input is gen-

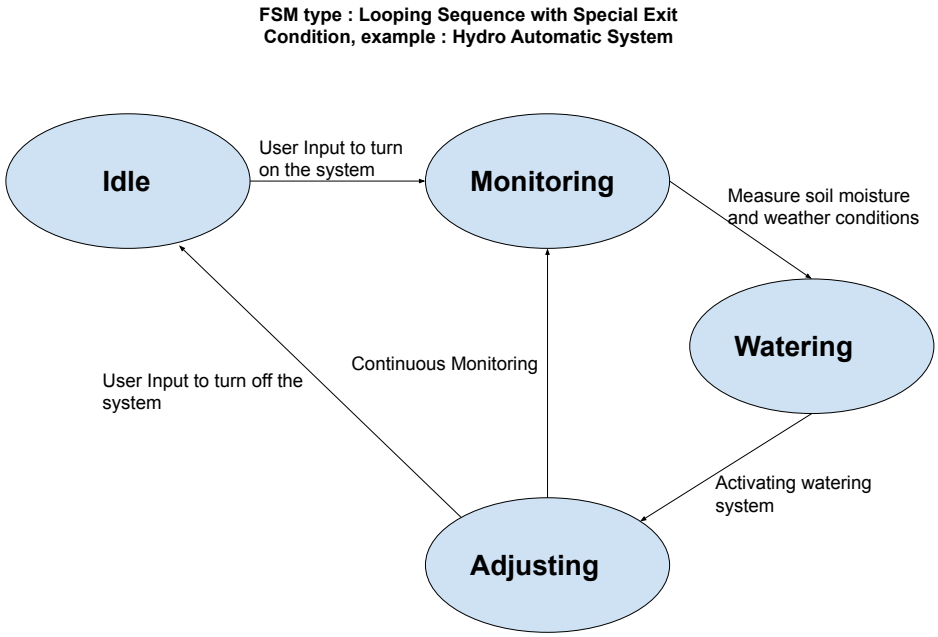


Figure 2.6: Example of Looping with Special Exit Condition FSM

erated randomly with a size of 64 symbols, if we unlocked all the states of FSM, again move to the first state. To generate a special condition, randomly choose a symbol and state count, if ever hit that condition, stop the execution else keep on moving in the loop.

5. **Arbitrary FSM**- All the Finite State machines that *doesn't follow just one fixed pattern* but include some or all the above FSMs fall into this category. In this, we randomly decide which pattern the FSM will follow and for how many symbols and keeps on doing this till we reach the desired number of states.

   For example, in a payment system, first, we enter credit card details, it should follow Type 2 (Special sequence with a reset after a mismatch), the next sequence is adding Pin details which should also follow the previous same pattern and last method is adding Billing Address where it is allowed to have some mismatch in the symbols hence follow Type 3 (Reset after a few miss).

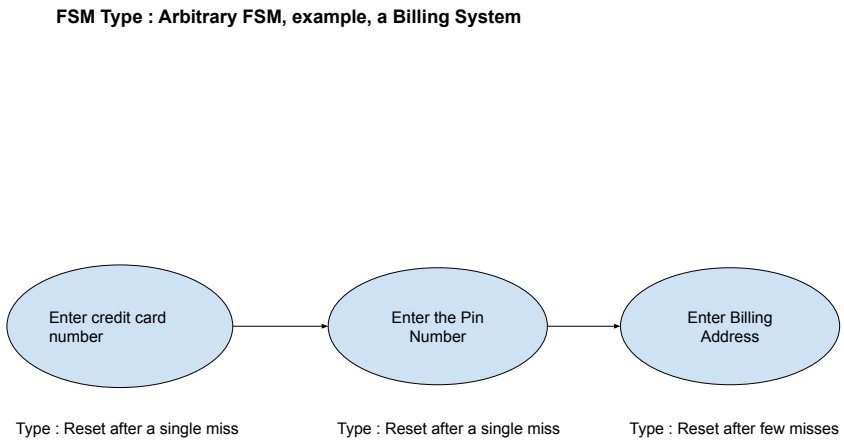


Figure 2.7: Example of Arbitrary FSM

For this type of FSM, we fix the output FSM symbols. For the input symbol, we randomly choose a type and the number of symbols, and we generate the input accordingly. We keep on generating new seeds till we don't find the match. After we have the match,

6. **Non-Deteministic FSM** - All the FSM whose behavior is *not deterministic and depends on variables such as Action, Time, environment conditions, etc* fall into this category. *For example,* Uncertainty in Sensor Readings - industrial Control Systems often rely on sensor readings to make control decisions. However, noise, drift, or other sources of uncertainty can affect sensor readings, leading to non-deterministic behavior. For example. consider a control system that regulates the pressure of a process sensor reading. Since it is affected by noise, the control system may make different control decisions for the same pressure setpoint.

Since all the Non-Deterministic FSM can be represented with the above categories, we did

not implement this FSM as another type.

2.2.3 Test Case Generation

For test case generation, we use three methods -

1. **Random Seed Generator** - *To generate test cases*, we use the functionality of a random number generator along with the string functionality. This algorithm takes two arguments (number of states in FSM and size of each state) and generates an input buffer seed of the fixed length of each symbol and the fixed length of the total number of symbols. We use this API to generate both the input seed buffer and the output FSMs. For all the categories of FSMs, we use the same function to generate the Finite State Machines input.

2. **Minimisation Across Seeds** - The goal of this algorithm is to *create a buffer of seeds whose union is 100% test coverage.* Instead of randomly choosing input seeds for exploration, this algorithm finds the seeds which guarantee some match with the desired state. We take the input seed and find a match with the desired state if the match includes discovering new states and the count of the new state is greater than the threshold, we store the input seed in the buffer and keep on doing this till the test coverage is 100. While choosing input for exploration, we randomly choose input from this buffer instead of taking any random input.

3. **Minimisation Within Seeds** - The goal of this algorithm is to find the *minimum number of symbols required to have a perfect match with the desired states.* We remove the contiguous same symbols and keep the output as concise as possible. The symbols are modified by comparing them with the output states of the FSM.

2.2.4 FSM Algorithm Modifications for Feedback Integration

We implemented 5 different categories of Finite State Machines. For each category, we try to find the perfect match of the input seed with the buffer but if we failed to do we discard the input and generate a new random seed. We keep on doing this till we don't get the perfect match.

For an input seed, we might have some good symbols that we are discarding completely if we don't get a perfect match with the output. For FiniteFuzz, we capture good symbols that are symbols that increase feedback and explore the remaining symbols. In this way, we are not wasting exploration by generating the same symbols again and again.

We modified all 5 algorithms of different FSMs to include the feedback, we were able to unlock the Finite State Machine in linear time whereas the random input generator took exponential time for the same states.

Algorithm 1: FiniteFuzz Algorithm

Input: symbolsCount: Number of states in FSM, symbolSize: Length of each state

Output: seedsRequired: Number of seeds required to generate FSM

```
j ← i
outputFSM ← rand(symbolsCount, symbolSize)
inputFSM ← rand(symbolsCount, symbolSize)
matchedSymbols ← ""
matchedSymbolsCount ← 0
seedsRequired ← 0
while matchedSymbolsCount < symbolsCount do
    currMatchedCount ←
      matched symbols between inputFSM and outputFSM
    if currMatchedCount > prevMatchedCount then
        inputFSM ← modify input to add new coverage
    if currMatchedCount == prevMatchedCount then
        inputFSM ← randomly generate rest of the symbols
    seedsRequired ← seedsRequired + 1
    prevMatchedCount ← matchedSymbolsCount
    matchedSymbolsCount ← currMatchedCount
```

Figure 2.8: FiniteFuzz: Feedback-Guided Grey Box Fuzzing Algorithm for Finite State Machine Testing

2.2.5 Fuzzing Execution

For the first step, we have two options, either generate a random seed input and use this input or generate a buffer of random inputs such that the union of these buffer inputs gives 100 percent test coverage and use any random input from the buffer.

In the second step, we compare the input seed with the target FSM, as we are comparing different states of FSM, we keep on capturing the matched symbols.

In the third step, we are done with the matching phase, we compare the previously matched symbols and currently matched symbols, if the currently matched symbols are greater than the previous ones, we add the new coverage to the existing coverage. If no new coverage we discard the input seed.

We go back to step 1 till we don't unlock the FSM that is we keep on executing the above steps in the same order till we don't have 100% coverage.

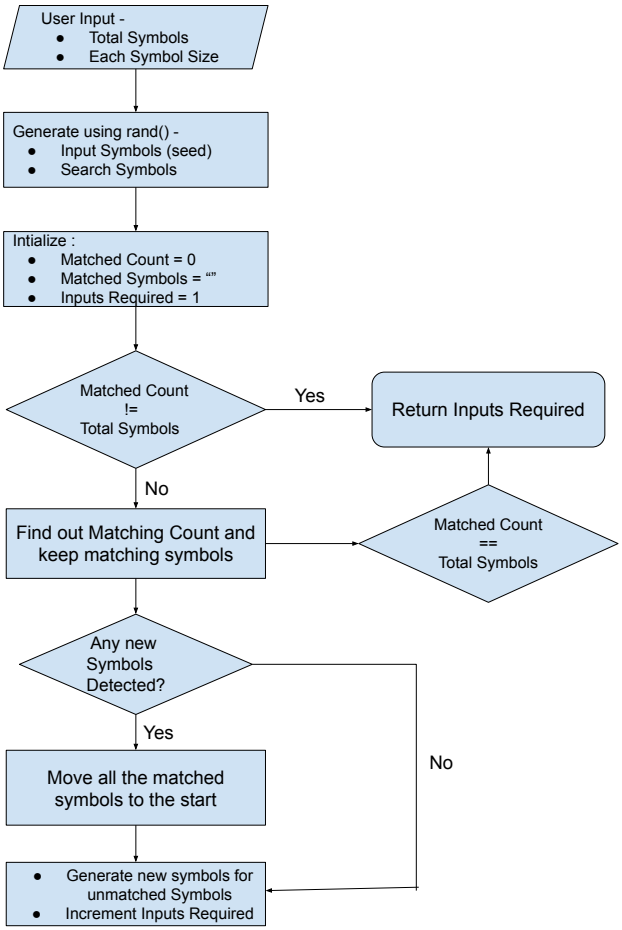


Figure 2.9: From Inputs to Results: The Grey Box Fuzzer Mutation Algorithm in a Nutshell

Chapter 3

Results

3.1 Evaluation

Three fundamental research questions guide our evaluation of the effectiveness of Finite Fuzzer -

1. **Evaluation Ques:** How to test the Finite State Machines?
2. **Evaluation Ques:** How does incorporating feedback into testing improve the efficiency of the testing?
3. **Evaluation Ques:** Is there any other way to increase the testing speed apart from adding feedback? Can we improvise the input as well?

We first get the length of the Finite State Machine from the user and the number of symbols in each state. We then pass these inputs to the random input generator function, which generates a fixed-length Finite State Machine and fix symbols in each state. For example, if the user gives the length of a Finite State Machine as 5 and the length of each symbol as 3, a random input generator will generate an input like this ”010 110 010 000 110”. In this way, we randomly decide our output desired state.

3.1.1 RQ1: How to test the Finite State Machines?

Now the big question is how to test the Finite State Machines. Which FSM to test? How to ensure we have covered all the scenarios? To answer all these questions, we thoroughly analyzed different Finite State Machines and divide them into 5 different categories -

1. **Monotonic Forward** The Finite State Machine that stays in the same state for the invalid state and only moves forward when a valid state is encountered falls into this category.For example, the Countdown timer(moves forward from starting time to zero and then resets when time ends, and Automatic Coffee Machines (moves in three states - Heating Water, Brewing Coffee, and keeping it warm).

2. **Special Sequence with Reset on Miss** For all the Finite State Machines that need continuous correct input to fall into this category, any wrong input will reset the state of the Finite State Machine. For example, Security System uses FSM to control access to a restricted area. The input signal could be the presentation of a valid identification card and the entry of the correct password; the output would be the door opening. If the sequence is not contiguous, the security system could reset to its original state. An ATM also falls into the same category. When the system is idle, it is waiting for a user to insert their card and enter the pin. When the user adds the card and the PIN, the machine transition through the input state and verifies the information. If the information is invalid, the machine return to its idle state.

3. **Special Sequence with Reset after fix misses** There are Finite State Machines where a certain degree of error is allowed and if we exceed that threshold, the FSM resets. The threshold can vary depending on the task the user trying to perform. There are Finite State Machines where we are allowed to have a few mistakes and if we exceeded those

mistakes, it will reset the sequence.

4. **Looping Sequence with Special Exit** All the Finite State Machines that keep on looping in a circle fall into this category. They exit with a special condition and this special condition can either be deterministic or non-deterministic. An example of this category includes Traffic Light Signals (Non-Deterministic, keeping on Looping in three states - Red, Yellow, and Green and exit only if there is any issue in the sensor). Another example that covers Deterministic Behavior is Automatic Toll Booth, it has finite states such as Idle, Collecting, and Finished. When the toll is idle, it is waiting for the car to approach, and the toll booth transitions to the collecting state and shuts off. If another vehicle approaches, it follows the same process. The special exit condition in this case would be the toll booth running out of change or paper.

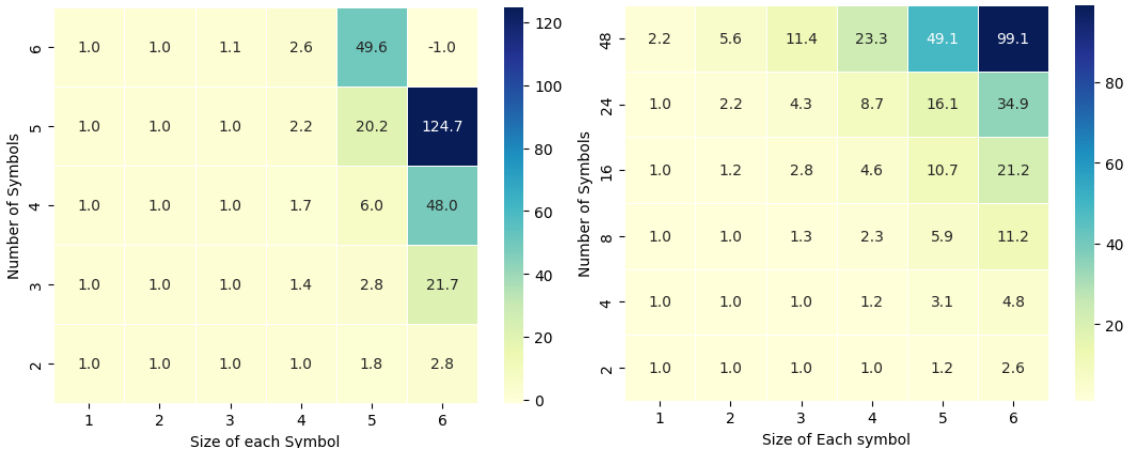
5. **Non Deterministic transition (Action, time)** The finite machines whose behavior cannot be determined to fall into this category. The main reasons are changes in the hardware behavior (sensor readings) and environmental conditions such as noise or temperature. For example, a control system that regulates the pressure of a process is based on a pressure sensor reading. If it is affected by noise, the control system may make different control decisions for the same pressure setpoint.

   The Finite State Machine that uses a combination of two or more of the above categories lies in this domain. The output of one type of FSM is the input of the next states and so on. Any FSM will fall into one of these categories. We implemented all these Finite State Machines categories to compare with the Black box fuzzer and the Finite Fuzzer efficiency.

3.1.2 RQ2: How does incorporating feedback into testing improve the efficiency of the testing?

To check how efficient the FiniteFuzz is in finding the desired state of FSM, we conducted the experiments in a Black-box fuzzer for all the above 5 FSM types and compared them with the FiniteFuzz. The results of these FSM are as follows -

1. **Monotonic Forward** We ran the Black-box fuzzer with states varying from 2 to 6 with the size of each state varying from 1 to 6. After increasing the state count from 6, it was taking a long time to reach the end state of FSM. As the number of states in FSM increases, the number of inputs required to reach the end state become exponential. For example, for 4 states with each symbol size as 6 - "010100 010101 011111 110101", to generate this state, the Black Box fuzzer took an average of 48 inputs and this is the max size we can get with a finite number of inputs with Black box fuzzer. With the FiniteFuzz, we were easily able to generate a high number of states. We limit the number of states to 48 and kept the variation in the size of each state from 1 to 6 which is the same as the Black-box fuzzer. Here we saw a linear increase in the inputs required compared to Black Box Fuzzer which had an exponential performance. Comparing the same 4 states with the size of each state as 6, we were able to generate FSM in an average of 4.8 inputs. The

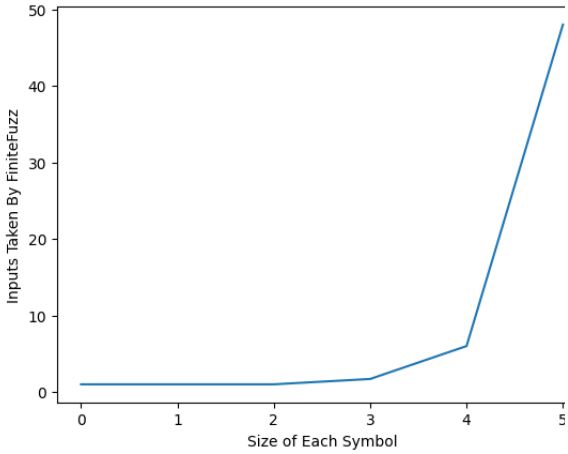


(a) BlackBoxFuzzer for Monotonic FSM

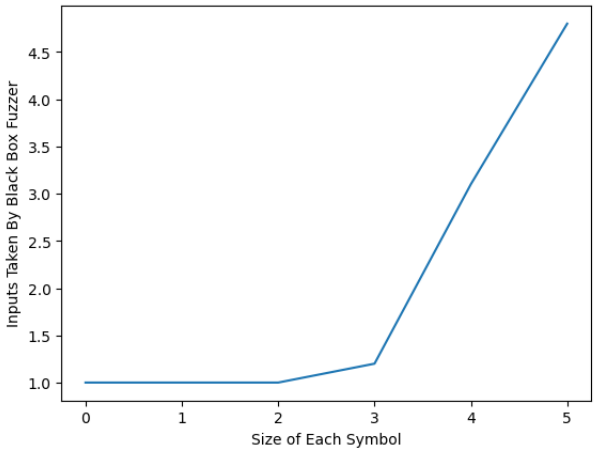
(b) FiniteFuzz for Monotonic FSM

Figure 3.1: Effectiveness Comparison of BlackBoxFuzzer and FiniteFuzz for Monotonic FSM. This heat map compares the bug detection effectiveness for FSM of Type Arbitray FSM of black box fuzzer and FiniteFuzz. The x-axis represents the size of each symbol while the y-axis represents the number of states. Darker colors indicate higher input count and longer testing times. The data suggest that FiniteFuzz is more effective than BlackBoxFuzzing for bug detection.

compares the heat map of different numbers of states. The



(a) BlackBoxFuzzer with Monotonic FSM

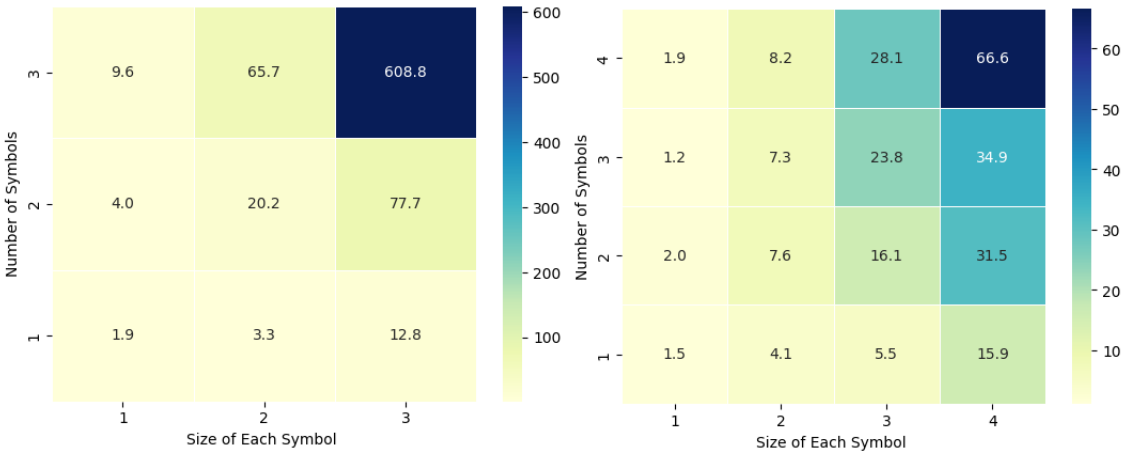


(b) FiniteFuzz for Monotonic FSM

Figure 3.2: Comparison of Input Count Vs Symbol Size for BlackBoxFuzzer and FiniteFuzz for Monotonic FSM: This graph compares the two fuzzers on the basis of Input and Symbols Size, the x-axis represents the size of each generated symbol while the y-axis represents the number of inputs taken by each fuzzer. The results show the efficiency of FSM generation for each fuzzer, with the FiniteFuzz demonstrating a more consistent trend of generating efficient FSM with the increasing symbol size.

compares the number of inputs where the states are fixed to 4 and the size of each symbol varies from 1 to 5. An average number of 5 inputs is required for the maximum case with the FiniteFuzz and a maximum of 50 inputs with the Black-box fuzzer. The FiniteFuzz for this FSM is at **least 10X faster** in generating FSM compared to the Black-box fuzzer.

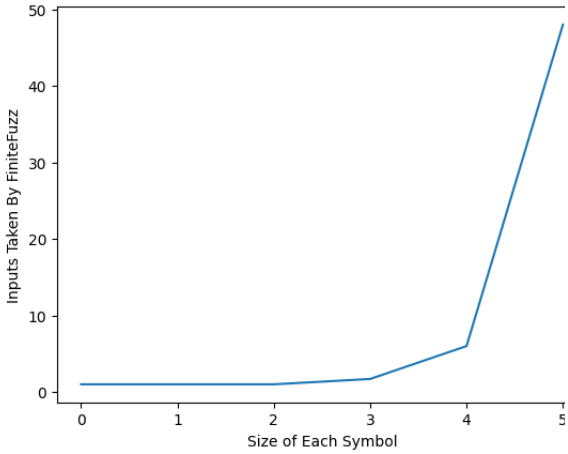
2. **Special Sequence with Reset on Miss** Since in this type of FSM, any wrong symbol leads to a reset, we could not generate an FSM for a higher number of states, as the number of symbols and each symbol size keeps on increasing, the number of inputs became exponential very quickly. In a finite amount of time, we were able to generate input with a maximum of 3 states and a symbol size of 3. For 3 states with a symbol size of 3, for example, for the desired state "010 101 111", it took an average of 608.8 inputs to reach the desired state. For FiniteFuzz, we were able to achieve this state with an average of 24 inputs. Also, we were easily able to generate the FSM for a higher number of states but we limit it to a maximum of 4 states with each state varying from 1 to 4. We have linear time performance with the FiniteFuzz and exponential performance with the Black-box fuzzer. The heatmap compares the number of inputs for a different number of states and the size of each state in



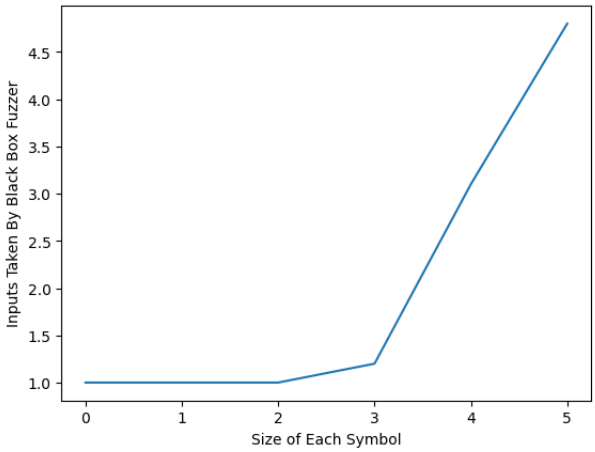
(a) BlackBoxFuzzer for Reset On Miss FSM

(b) FiniteFuzz for Reset on Miss FSM

Figure 3.3: Effectiveness Comparison of BlackBoxFuzzer and FiniteFuzz for FSM with reset on miss - This heat map compares the bug detection effectiveness for FSM of Type Arbitray FSM of black box fuzzer and FiniteFuzz. The x-axis represents the size of each symbol while the y-axis represents the number of states. Darker colors indicate higher input count and longer testing times. The data suggest that FiniteFuzz is more effective than BlackBoxFuzzing for bug detection.



(a) BlackBoxFuzzer for Reset On Miss FSM



(b) FiniteFuzz for Reset on Miss FSM

Figure 3.4: Comparison of Input Count Vs Symbol Size for BlackBoxFuzzer and FiniteFuzz for Reset On FSM: This graph compares the two fuzzers on the basis of Input and Symbols Size, the x-axis represents the size of each generated symbol while the y-axis represents the number of inputs taken by each fuzzer. The results show the efficiency of FSM generation for each fuzzer, with the FiniteFuzz demonstrating a more consistent trend of generating efficient FSM with the increasing symbol size.

compares the number of inputs where the states are fixed to 3 and the size of each symbol varies from 1 to 3. An average number of 35 inputs is required for the maximum case with the FiniteFuzz and a maximum of 600 inputs with the Black-box fuzzer. We have seen an **improvement of around 25X** for this FSM type.

3. **Looping Sequence with Special Exit** Since the system can exit early, it performs better than the previous two types, the behavior within the symbol is exponential but we were able to generate FSM for a higher number of states compared with the first two.

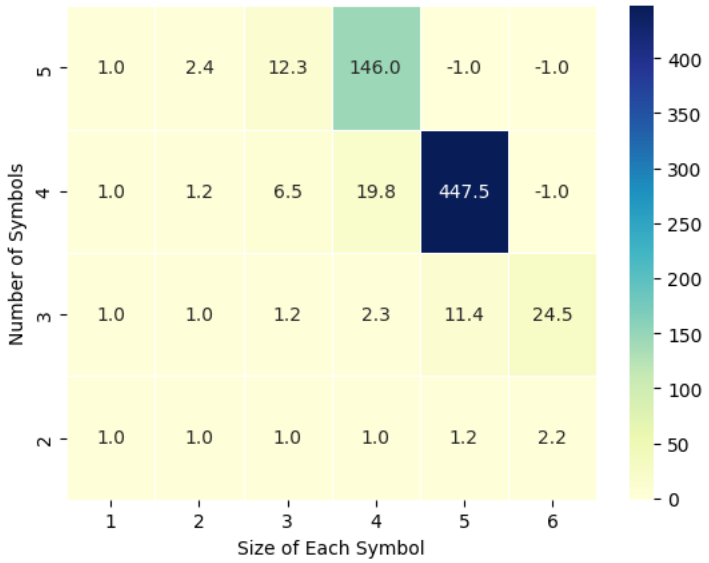


Figure 3.5: Black Box Fuzzer HeatMap for Looping with Special Exit Condition

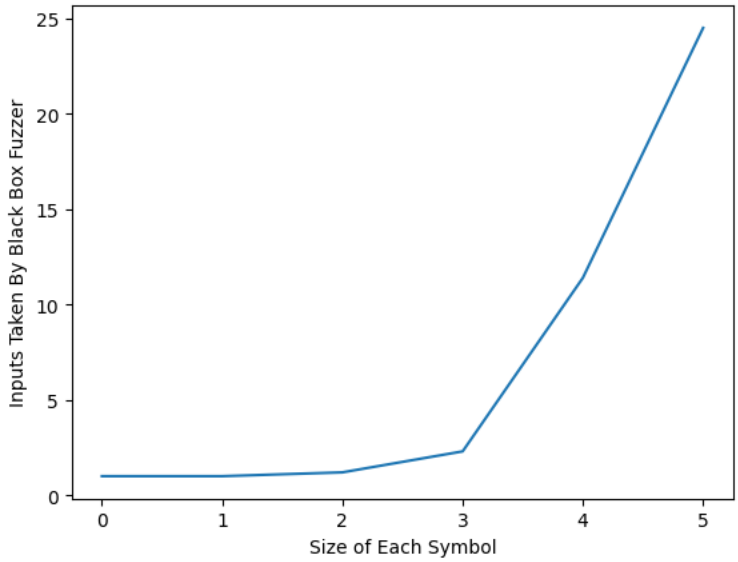
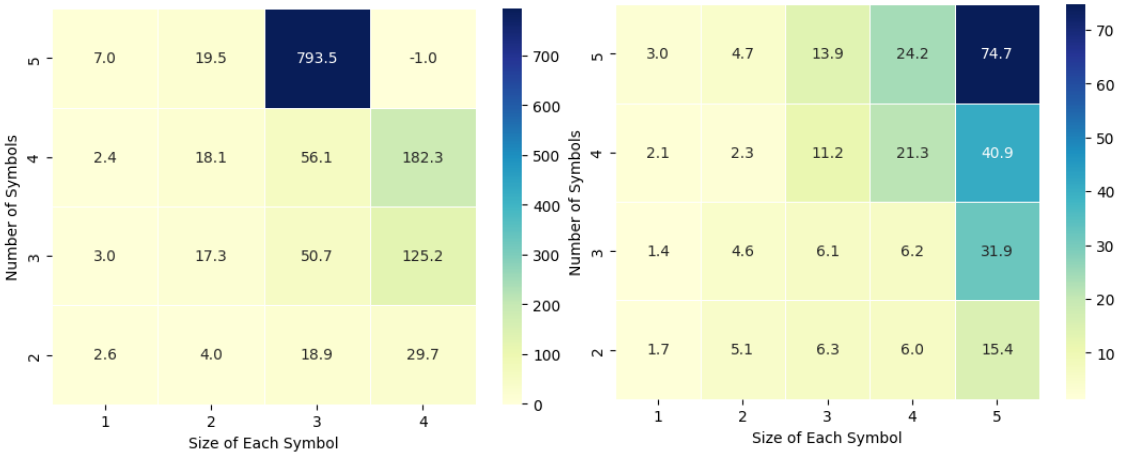


Figure 3.6: Black Box Fuzzer Graph for Looping with Special Exit Condition

4. **Combined DFA** Since it has a combination of all the above FSM, it outperforms better than the above types. Lesser number of inputs is required to generate FSM of a particular length. In this type of Finite State Machine, we were able to generate inputs for states varying from 2 to 5 with symbol sizes varying between 1 to 4 in a finite amount of time. For a Finite State Machine, ”010 110 001 111” if we have any random combination of the above types, it takes an average of 56 symbols to reach the final state for a Black box fuzzer. For the same input, the FiniteFuzzer takes an average of 11 inputs. We see a huge improvement in reaching the final stage. The black box fuzzer generates input in exponential time and fails if the number of states is greater than 5 whereas the Grey Box Fuzzer works in linear time and can generate input for a higher number of states as well.



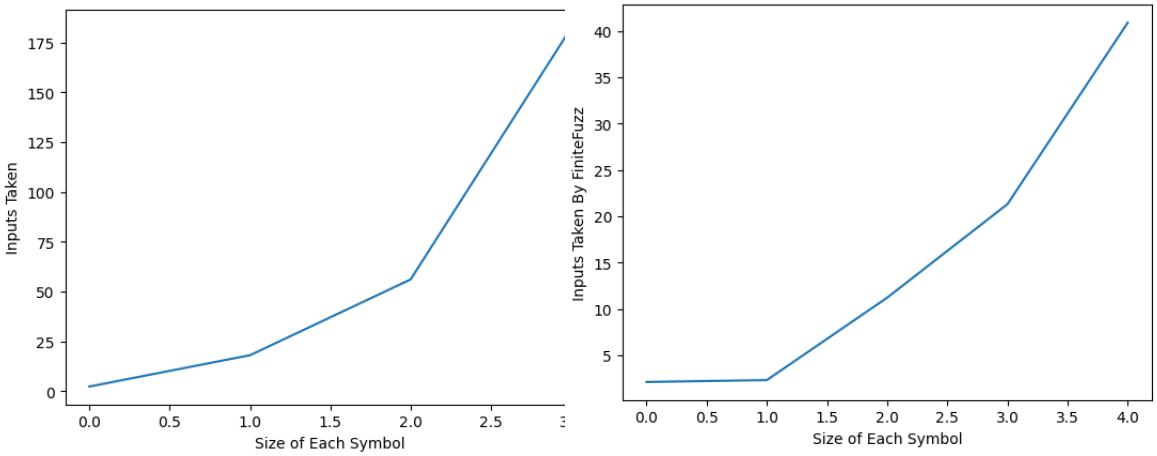
(a) BlackBoxFuzzer for Arbitray FSM

(b) FiniteFuzz for Arbitray FSM

Figure 3.7: Effectiveness Comparison of BlackBoxFuzzer and FiniteFuzz for ArbitraryFSM - This heat map compares the bug detection effectiveness for FSM of Type Arbitray FSM of black box fuzzer and FiniteFuzz. The x-axis represents the size of each symbol while the y-axis represents the number of states. Darker colors indicate higher input count and longer testing times. The data suggest that FiniteFuzz is more effective than BlackBoxFuzzing for bug detection.

we see an improvement of at least 5X with the FiniteFuzz if we compare with the Black-box fuzzer.

With the feedback, we have seen an outstanding improvement in the results, the FSMs that were taking exponential time to reach a desired state have now moved to linear time. Here is the difference for all the FSM types.



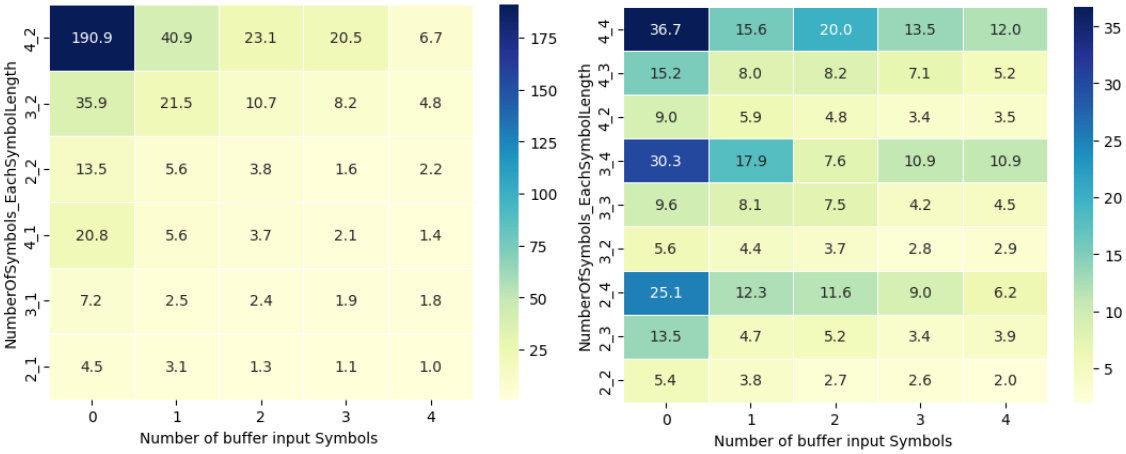
(a) BlackBoxFuzzer for Arbitrary FSM

(b) FiniteFuzz for Arbitrary FSM

Figure 3.8: Comparison of Input Count Vs Symbol Size for BlackBoxFuzzer and FiniteFuzz for Arbitrary FSM: This graph compares the two fuzzers on the basis of Input and Symbols Size, the x-axis represents the size of each generated symbol while the y-axis represents the number of inputs taken by each fuzzer. The results show the efficiency of FSM generation for each fuzzer, with the FiniteFuzz demonstrating a more consistent trend of generating efficient FSM with the increasing symbol size.

. The FiniteFuzz for this FSM is **5X better compared with Black Box Fuzzer**.

5. **Special Sequence with Reset after fix misses-** For the Black-box fuzzer, we were able to generate inputs with states varying from 2 to 4 and the size of each state varying from 1 to 2. We kept the threshold from 0 to 4 extra states. For example, if we are trying to find an FSM that looks like "0100 1100 0000 1010" and 1 miss is allowed, any of the following can be a valid input - "0100 1100 0000 0100 1010", "0100 1100 1000 0100 1010", "1010 1100 0000 0100 1010", etc. are valid inputs. For an FSM, where 2 skips are allowed, the number of states is 4 and the size of each symbol is 2, "10 01 01 11", we need 23 input symbols. With the FiniteFuzz we were able to generate FSM for a number of states varying from 2 to 4 and the size of each state varies from 2 to 4. We kept the same number of symbols from 0 to 4. With the same FSM, 4 states, and size of each state as 2 and buffer symbols as 2, FiniteFuzz took an average of 4.8 inputs.

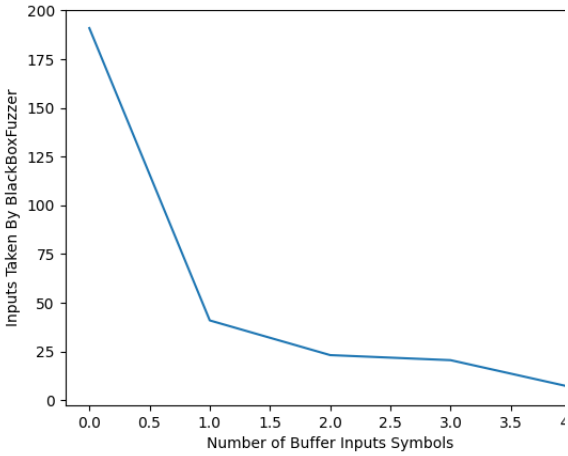


(a) BlackBoxFuzzer for FSM with Skips

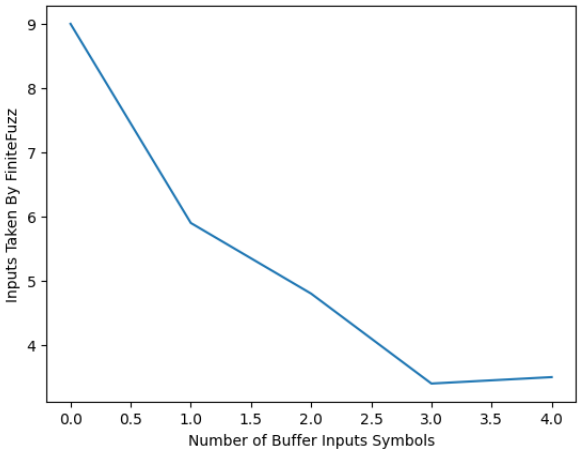
(b) FiniteFuzz for FSM with Skips

Figure 3.9: Effectiveness Comparison of BlackBoxFuzzer and FiniteFuzz for FSM with Skips - This heat map compares the bug detection effectiveness for FSM of Type Arbitray FSM of black box fuzzer and FiniteFuzz. The x-axis represents the size of each symbol while the y-axis represents the number of states. Darker colors indicate higher input count and longer testing times. The data suggest that FiniteFuzz is more effective than BlackBoxFuzzing for bug detection.

compares the heatmap of the FiniteFuzz and the Black-box fuzzer.



(a) BlackBoxFuzzer for FSM with Skips



(b) FiniteFuzz for FSM with Skips

Figure 3.10: Comparison of Input Count Vs Symbol Size for BlackBoxFuzzer and FiniteFuzz for FSM with Skips: This graph compares the two fuzzers on the basis of Input and Symbols Size, the x-axis represents the size of each generated symbol while the y-axis represents the number of inputs taken by each fuzzer. The results show the efficiency of FSM generation for each fuzzer, with the FiniteFuzz demonstrating a more consistent trend of generating efficient FSM with the increasing symbol size.

compares the number of inputs required to generate FSM when the number of states is 4 and the size of each symbol is 2 with the threshold of 2 error symbols. The FiniteFuzz for this FSM is **25X better compared with Black Box Fuzzer**.

3.1.3 RQ3: Is there any other way to increase the testing speed apart from adding feedback? Can we improvise the input as well?

In the Black Box Fuzzer, we are completely dependent on the random inputs, in the Grey Box Fuzzer, we are dependent on the random inputs but we were capturing the good states as well which is acting as the feedback. Now, the question is can we improvise on these random inputs? To explore this, we implemented two algorithms -

1. **Minimisation Across Seeds -** We need a good selection of samples, it's not totally dumb fuzzing, we need a good selection of sample. We can have infinite possibilities of starting symbols for starting seeds, we can run with min set of code coverage of these documents. For example, with any seed, we need 5% code coverage and we run another file, we get 5% code coverage but with a little bit more in some other way, what we do is try to generate the minimum set of necessary files so that we maximize the unique new code blocks that we hit and throw the other 10, 000 files that were not giving the new coverage and focus on just special 800 of those that give that coverage.

   We have devised a novel seed minimization algorithm. The core idea of my algorithm is to assemble a buffer of input seeds that provides complete test coverage with no redundancy. To achieve this, I have designed the algorithm to ensure that the union of all seeds in the buffer covers 100[%] of the code paths, without any overlap in coverage between individual seeds. The seeds are added in such a way that it always increases the coverage of the

Finite State Machine. By selecting seeds from this buffer, we can rapidly construct a Finite State Machine that provides a comprehensive view of the system's behavior under different inputs.

One of the primary benefits of my algorithm is its ability to systematically explore the input space and identify potential vulnerabilities. Because the seeds in the buffer provide complete test coverage with no overlap, each new seed selected from the buffer is guaranteed to expand the test coverage of the system. This means that we can be sure that we are not missing any critical inputs that could trigger a vulnerability. Additionally, the efficiency of the algorithm means that we can explore a large number of inputs in a relatively short period of time, further improving the likelihood of identifying potential issues.

2. **Minimisation Within Seeds -** The goal of this algorithm is to find the minimum number of symbols required to have a perfect match with the desired states. We remove the contiguous same symbols and keep the output as concise as possible. The symbols are modified by comparing them with the output states of the FSM.

   After we unlock an FSM, we may end up with symbols that are redundant. For example, for Montonic FSM, if the targeted FSM is "010 000 101" and the input that unlocks FSM is "010 000 000 000 010 101", we can minimize this input FSM as the states "000" gets repeated more than required. After we have done with the minimization, we will end up with input FSM as"010 000 010 101". It helps to reduce redundant spaces.

Chapter 4

Discussion

We started this research with the goal of increasing the speed of generating a Finite State Machine. We were able to categorize the FSM into different categories and implement each category. To compare the improvement we achieved, we first ran all the categories with the Black-Box fuzzer and then with the /tool. For each category, we saw a significant improvement in the number of inputs required to generate the Finite State Machine. For instance, for Monotonic FSM, we saw a **10X increase in the speed**, and for Special Sequence with Reset on Miss the **increase in speed is around 25X**, for combined DFA, the **increase in speed is around 5X** and for FSM where we have a certain threshold, **the increase in speed is 5X**.

The key advantage of our approach is that it reduces the number of seed inputs needed to generate the FSM. Unlike the traditional black box fuzzing techniques which require an exponential number of see inputs based on the number of states in the FSM, our Grey box fuzzer uses feedback to go from one state to another, reducing the number of seed inputs needed.

Going a step further, we implemented minimization algorithms as well. Since the state space can be huge, we are successfully able to implement an algorithm that minimizes this state space and only contains inputs that have new state coverage and no overlapping coverage. This can further reduce the number of inputs. Also, we implemented Minimisation within the seeds algorithm that can help us in exploration and exploitation and avoids looping over a state again and again, saving time and space for seed storage.

One limitation of our approach is that it assumes that each state will have a fixed size and explore the feedback accordingly. Future work will involve exploring different sizes for each state in a single Finite State Machine.

In conclusion, our FiniteFuzz represents a significant improvement over traditional Black Box FSM fuzzing techniques. Its ability to generate test cases based on feedback makes it a valuable tool for software testing and security analysis.

Chapter 5

Conclusions

In this research, we divided Finite State Machines into different categories and implemented each category. We generated the Finite State Machines with Black Box Fuzzer and FiniteFuzz, and we have seen tremendous improvement with FiniteFuzz for all the categories in generating the Finite State Machines. The FSMs which were taking exponential inputs with Black Box Fuzzer are now able to generate the same FSM with linear input seeds.

The next plan of action is to use the FiniteFuzz for the real-world applications of Finite State Machines. The plan is to first start using different protocol implementations. Might require adding a handshaking mechanism for using FiniteFuzz for protocol fuzzing. Regex is another important application of Finite State Machines and it is used widely in the IDEs, command lines, databases, etc. [30] discuss the problems in regex such as security vulnerabilities, testing coverage, etc, and the nature of issues in regular expressions. we will use the FiniteFuzz to address these vulnerabilities and try to uncover more vulnerabilities.

Bibliography

[1] Mohiuddin Ahmed, Abdun Naser Mahmood, and Jiankun Hu. A survey of network anomaly detection techniques. *Journal of Network and Computer Applications*, 60: 19–31, 2016.

[2] Michael Barr. How programmable logic works. *Embedded Systems Programming*, pages 75–84, 1999.

[3] Marcel Böhme, Cristian Cadar, and Abhik Roychoudhury. Fuzzing: Challenges and reflections. *IEEE Softw.*, 38(3):79–86, 2021.

[4] Ryan Coppa, G Foudree, and D Greve. Fuzzm: A model-based approach to grey-box fuzzing. *Rockwell Collins, Tech. Rep.*, 2018.

[5] Grant Dick and Xin Yao. Model representation and cooperative coevolution for finite-state machine evolution. In *2014 IEEE Congress on Evolutionary Computation (CEC)*, pages 2700–2707. IEEE, 2014.

[6] Andre Drumea and Camelia Popescu. Finite state machines and their applications in software for industrial control. In *27th International Spring Seminar on Electronics Technology: Meeting the Challenges of Electronics Technology Progress, 2004.*, volume 1, pages 25–29. IEEE, 2004.

[7] Rong Fan, Jianfeng Pan, and Shaomang Huang. Arm-afl: coverage-guided fuzzing framework for arm-based iot devices. In *Applied Cryptography and Network Security Workshops: ACNS 2020 Satellite Workshops, AIBlock, AIHWS, AIoTS, Cloud S&P, SCI, SecMT, and SiMLA, Rome, Italy, October 19–22, 2020, Proceedings 18*, pages 239–254. Springer, 2020.

[8] Zicong Gao, Weiyu Dong, Rui Chang, and Yisen Wang. Fw-fuzz: A code coverage-guided fuzzing framework for network protocols on firmware. *Concurrency and Computation: Practice and Experience*, 34(16):e5756, 2022.

[9] Alasdair Gilchrist. *Industry 4.0: the industrial internet of things*. Springer, 2016.

[10] Anil K Gupta, Ken G Smith, and Christina E Shalley. The interplay between exploration and exploitation. *Academy of management journal*, 49(4):693–706, 2006.

[11] Mika Kasslin, Jari Kangas, and Olli Simula. Process state monitoring using self-organizing maps. In *Artificial neural networks*, pages 1531–1534. Elsevier, 1992.

[12] David Lee and Mihalis Yannakakis. Testing finite-state machines: State identification and verification. *IEEE Transactions on computers*, 43(3):306–320, 1994.

[13] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. Fuzzing: State of the art. *IEEE Transactions on Reliability*, 67(3):1199–1218, 2018.

[14] Wolfgang Mahnke, Stefan-Helmut Leitner, and Matthias Damm. *OPC unified architecture*. Springer Science & Business Media, 2009.

[15] Richard McNally, Ken Yiu, Duncan Grove, and Damien Gerhardy. Fuzzing: the state of the art. 2012.

[16] Charlie Miller, Zachary NJ Peterson, et al. Analysis of mutation and generation-based fuzzing. *Independent Security Evaluators, Tech. Rep*, 4, 2007.

[17] Paul Morrissey, Nigel P Smart, and Bogdan Warinschi. A modular security analysis of the tls handshake protocol. In *Advances in Cryptology-ASIACRYPT 2008: 14th International Conference on the Theory and Application of Cryptology and Information Security, Melbourne, Australia, December 7-11, 2008. Proceedings 14*, pages 55–73. Springer, 2008.

[18] Soukaina Najihi, Sakina Elhadi, Rachida Ait Abdelouahid, and Abdelaziz Marzak. Software testing from an agile and traditional view. *Procedia Computer Science*, 203: 775–782, 2022.

[19] Matthias Niedermaier, Florian Fischer, and Alexander von Bodisco. Propfuzz—an it-security fuzzing framework for proprietary ics protocols. In *2017 International conference on applied electronics (AE)*, pages 1–4. IEEE, 2017.

[20] Marcelo T Okano. Iot and industry 4.0: the industrial new revolution. In *International Conference on Management and Information Systems*, volume 25, page 26, 2017.

[21] Yan Pan, Wei Lin, Liang Jiao, and Yuefei Zhu. Model-based grey-box fuzzing of network protocols. *Security and Communication Networks*, 2022, 2022.

[22] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Aflnet: a greybox fuzzer for network protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 460–465. IEEE, 2020.

[23] Zhan Shu and Guanhua Yan. Iotinfer: Automated blackbox fuzz testing of iot network protocols guided by finite state machine inference. *IEEE Internet of Things Journal*, 9(22):22737–22751, 2022.

[24] Keith Stouffer, Joe Falco, Karen Scarfone, et al. Guide to industrial control systems (ics) security. *NIST special publication*, 800(82):16–16, 2011.

[25] Andrew S Tanenbaum. Network protocols. *ACM Computing Surveys (CSUR)*, 13 (4):453–489, 1981.

[26] Vaibbhav Taraate. Fsm designs. In *Digital Design from the VLSI Perspective: Concepts for VLSI Beginners*, pages 177–191. Springer, 2022.

[27] Victor L Voydock and Stephen T Kent. Security mechanisms in high-level network protocols. *ACM Computing Surveys (CSUR)*, 15(2):135–171, 1983.

[28] Artemios G Voyiatzis, Konstantinos Katsigiannis, and Stavros Koubias. A modbus/tcp fuzzer for testing internetworked industrial systems. In *2015 IEEE 20th conference on emerging technologies & factory automation (ETFA)*, pages 1–6. IEEE, 2015.

[29] Ferdinand Wagner, Ruedi Schmuki, Thomas Wagner, and Peter Wolstenholme. *Modeling software with finite state machines: a practical approach*. CRC Press, 2006.

[30] Peipei Wang, Chris Brown, Jamie A Jennings, and Kathryn T Stolee. An empirical study on regular expression bugs. In *Proceedings of the 17th international conference on mining software repositories*, pages 103–113, 2020.

[31] Ting Wang, Qi Xiong, Haihui Gao, Yong Peng, Zhonghua Dai, and Shengwei Yi. Design and implementation of fuzzing technology for opc protocol. In *2013 Ninth International Conference on Intelligent Information Hiding and Multimedia Signal Processing*, pages 424–428. IEEE, 2013.

[32] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. Scheduling black-box mutational fuzzing. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 511–522, 2013.

[33] Qi Xiong, Hui Liu, Yuan Xu, Huayi Rao, Shengwei Yi, Baofeng Zhang, Wei Jia, and Hui Deng. A vulnerability detecting method for modbus-tcp based on smart fuzzing mechanism. In *2015 IEEE International conference on electro/information technology (EIT)*, pages 404–409. IEEE, 2015.