`Gurthang` - A Fuzzing Framework for Concurrent Network Servers

Connor W. Shugg

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 & Applications

Godmar Back, Chair

Matthew Hicks

Haining Wang

May 5th, 2022

Blacksburg, Virginia

Keywords: Fuzzing, Systems, Networking, Security, Testing

Copyright 2022, Connor W. Shugg

`Gurthang` - A Fuzzing Framework for Concurrent Network Servers

Connor W. Shugg

(ABSTRACT)

The emergence of Internet-connected technologies has given the world a vast number of services easily reachable from our computers and mobile devices. Web servers are one of the dominant types of computer programs that provide these services to the world by serving files and computations to connected users. Because of their accessibility and importance, web servers must be robust to avoid exploitation by hackers and other malicious users. **Fuzzing** is a software testing technique that seeks to discover bugs in computer programs in an automated fashion. However, most state-of-the-art fuzzing tools (**fuzzers**) are incapable of fuzzing web servers effectively, due to their reliance on network connections to receive input and other unique constraints they follow. Past research exists to remedy this situation, and while they have had success, certain drawbacks are introduced in the process.

To address this, we created `Gurthang`, a fuzzing framework that gives state-of-the-art fuzzers the ability to fuzz web servers easily, without having to modify source code, the web server's threading model, or fundamentally change the way a server behaves. We introduce novelty by providing the ability to establish and send data across multiple concurrent connections to the target web server in a single execution of a fuzzing campaign, thus opening the door to the discovery of concurrency-related bugs. We accomplish this through a novel file format and two shared libraries that harness existing state-of-the-art fuzzers.

We evaluated `Gurthang` by performing a research study at Virginia Tech that yielded 48 discovered bugs among 55 web servers written by students. Participants utilized `Gurthang` to integrate fuzzing into their software development process and discover bugs. In addition,

we evaluated `Gurthang` against Apache and Nginx, two real-world web servers. We did not discover any bugs on Apache or Nginx, but `Gurthang` successfully enabled us to fuzz them without needing to modify their source code. Our evaluations show `Gurthang` is capable of performing fuzz-testing on web servers and discovering real bugs.

Gurthang - A Fuzzing Framework for Concurrent Network Servers

Connor W. Shugg

(GENERAL AUDIENCE ABSTRACT)

The Internet is widely apparent in our everyday lives. Since its creation, a wide variety of technologies and critical infrastructures have become accessible via the Internet. While this accessibility is a great boon for many, it does not come without risk. Web servers are one of the dominant types of computer programs that make the Internet what it is today; they are responsible for transmitting web pages and other files to connected users, as well as performing important computations per the user's request. Like any computer program, web servers contain bugs that may lead to vulnerabilities if exploited by a malicious user (a hacker). Considering they are open to all via the Internet, it is critical to catch and fix as many bugs as possible during a web server's development. Certain tools, called **fuzzers**, have been created to test computer programs in an automated fashion to discover bugs (called **fuzzing**, or **fuzz-testing**), although many of these fuzzers lack the ability to effectively test web servers due to the specific constraints a web server must follow. Previous research exists to fix this problem, but certain drawbacks are introduced in the process.

To address this, we developed `Gurthang`, a fuzzing framework that gives state-of-the-art fuzzers the ability to test a variety web servers, while also fixing some of these drawbacks and introducing a novel technique to test the concurrency aspects of a web server. We evaluated `Gurthang` against several web servers through a research study at Virginia Tech in which participating students performed fuzz-testing on web servers they implemented for their coursework. We discovered 48 bugs across 55 web servers through this study. We also evaluated `Gurthang` against Apache and Nginx (two web servers frequently used in the real

world) and showed `Gurthang` is capable of fuzzing them without the need to modify their source code.

Dedication

I am blessed with lots of wonderful friends and family to which I dedicate this thesis. I'm especially grateful for my two loving parents, Chris and Beth Shugg, who have been my biggest supporters and the best parents I could possibly ask for. To my brother Ben and sister Katie, I am incredibly grateful for their never-ending support and love. To my friends I have made throughout my college studies, I thank for their support and for the memories I'll cherish forever. To the Marching Virginians and everyone in it (with a heavy emphasis on my friends in the trumpet section), I thank for an unforgettable and life-changing five years. Lastly, to the late J.R.R. Tolkien, I thank for creating a world to which I often find myself escaping, and for inspiring the name of this thesis.

Acknowledgments

I would like to thank my advisor, Dr. Back, for helping me to perform this research and complete my degree. His guidance has been a huge help over the past few years, both while completing my degree and working with him as a teaching assistant for CS 3214. I have taken many valuable lessons from our collaboration and am very grateful for the time he has devoted to my work.

I would also like to thank the rest of my committee, Dr. Hicks and Dr. Wang, for agreeing to provide feedback and insight on my research. I am especially grateful to Dr. Hicks for hosting a Systems Security Seminar on Fuzzing during my final semester.

I also must acknowledge the many students I was fortunate enough to work with during my time as a teaching assistant. I thank them for challenging me to be a good mentor and a reliable peer. I am also very grateful to those who participated in my research study to make this thesis possible.

Finally, I would like to thank Professor William McQuain for introducing me to computer systems and inspiring me to go into this field.

Contents

List of Figures **xii**

1 Introduction **1**

1.1 Challenges of Fuzzing Network Applications 2

1.2 Proposed Solution 5

1.3 Contributions Made 8

1.4 Thesis Roadmap 9

2 Background **10**

2.1 Fuzzing 10

2.1.1 The Origins of Fuzzing 10

2.1.2 Types of Fuzzers 11

2.1.3 Basic Blocks 13

2.1.4 LLVM 14

2.1.5 `AFL` and `AFL++` 14

2.1.6 Memory-Related Vulnerabilities 16

2.2 Networking Fundamentals 17

2.2.1 TCP - Transmission Control Protocol 17

2.2.2 HTTP - Hypertext Transfer Protocol 19

2.2.3 Web Server Concurrency Design 26

2.3 Operating Systems Fundamentals 27

2.3.1 Linux Standard File Streams 27

2.3.2 Linux Networking Sockets 28

2.3.3 Linux Shared Libraries & LD_PRELOAD Interposition 30

2.3.4 Linux Process Signals 32

3 Design and Implementation **34**

3.1 Design Overview 35

3.1.1 Bridging the Input Gap 37

3.1.2 The Comux File Format 37

3.1.3 The Comux Mutator 38

3.2 The Comux File Format 39

3.2.1 File Layout 39

3.2.2 The Comux Toolkit 42

3.3 The Gurthang LD_PRELOAD Library 42

3.3.1 Internal Threading 44

3.3.2 Connection Table 45

3.4 The Gurthang AFL++ Custom Mutator 46

3.4.1 `Comux` Inspection . . . 46
3.4.2 `Comux` Mutation Strategies . . . 47
3.4.3 Test Case Trimming . . . 52
3.5 Design Limitations . . . 53

4 Evaluation **55**
4.1 Evaluation Goals . . . 55
4.2 Research Study . . . 55
4.2.1 HTTP Server Project . . . 56
4.2.2 Study Protocol . . . 57
4.2.3 Study Participation . . . 60
4.2.4 Unit Testing Prior to Participation . . . 61
4.2.5 Assessment of `Gurthang`'s Use By Participants . . . 67
4.2.6 Examples of Bugs Discovered by `Gurthang` . . . 71
4.2.7 Participant Bug Fixes over Multiple Fuzzing Campaigns . . . 81
4.3 Survey Results . . . 87
4.4 Fuzzing Real-World Web Servers . . . 90
4.4.1 Fuzzing Apache . . . 91
4.4.2 Fuzzing Nginx . . . 94
4.5 Evaluation Results . . . 96

4.6 Evaluation Limitations 99

5 Related Work **101**

5.1 Related Work in Fuzzing 101

5.2 Network Application Fuzzers 104

6 Future Work **107**

6.1 Increased Performance 107

6.2 Protocol Awareness 108

6.3 True Parallel Communication 108

6.4 Further Testing of Real-World Web Servers 109

7 Conclusions **110**

Bibliography **112**

List of Figures

1.1 Challenge 1 - Input Management. 3

1.2 Challenge 2 - Fuzzing Across Multiple Connections of Varying Order. 4

1.3 Challenge 3 - Fuzzing Message Boundaries. 5

1.4 A high-level view of `Gurthang`'s architecture. 7

2.1 Depiction of a standard fuzzing loop. 11

2.2 An HTTP request message depicting a `GET` request for `/index.html`. 19

2.3 An HTTP request message depicting a simple authentication request. 20

2.4 An HTTP response message depicting a possible response for a `GET` request for `/index.html` 22

2.5 An HTTP response message depicting a possible response for a failed login attempt. 22

2.6 An HTTP request message depicting a simple authentication request. 25

2.7 An HTTP response message depicting the server's response to a successful login attempt. 25

2.8 An HTTP request message depicting the client choosing to use a previously-received cookie in order to gain access to a private file. 26

3.1 `Gurthang`'s logo, created by Connor Shugg. 34

3.2 A diagram depicting `Gurthang`'s architecture. 36

3.3 The `Comux` file architecture. 40

3.4 One example of a possible scheduling of four chunks in a single `comux` file. . . 41

3.5 A depiction of the internal workings of the `Gurthang` `LD_PRELOAD` library. . . 43

3.6 An example `comux` file with two connections and three chunks. 49

3.7 The same example `comux` file as shown in Figure 3.6, with a new scheduling value for chunk `C-0`. 50

4.1 The introduction screen displayed by the Python script to the participants. . 58

4.2 An example of a crash report displayed by the Python script to the participants after fuzzing concludes. 59

4.3 Occurrences of distinct servers submitted under separate participant IDs. . . 61

4.4 Overall average percentages of the unit tests' basic block coverage among all study participants. 63

4.5 A summary of the 55 distinct web servers submitted throughout the CS 3214 study. 66

4.6 The three bug categories determined by examining the 48 discovered bugs. . 66

4.8 Source functions from which `NULL` return values originated. 68

4.7 C code depicting bugs originating from the failure to check a return value. . 68

4.9 C code depicting a buffer overflow bug. 69

4.10 C code depicting a buffer underflow bug. 70

4.11 C code depicting an out-of-bounds memory read bug. 70

4.12 C code depicting an out-of-bounds memory read bug. 71

4.13 Example Bug 1 - The `comux` input file generated by `Gurthang`. 72

4.14 Example Bug 1 - The target web server crashes when given the input file. . . 73

4.15 Example Bug 1 - The web server's post mortem stacktrace (GDB). 74

4.16 Example Bug 1 - Examining the `NULL` pointer returned from a call to `strtok_r()`. 75

4.17 Example Bug 2 - The `comux` input file generated by `Gurthang`. 76

4.18 Example Bug 2 - The target web server crashes when given the input file. . . 76

4.19 Example Bug 2 - The web server's postmortem stacktrace (GDB). 77

4.20 Example Bug 2 - Viewing the parsed `Content-Length` header as -47 (GDB). 78

4.21 Example Bug 2 - Witnessing a stack-local variable be overwritten in an out-of-bounds write (GDB). 78

4.22 Example Bug 3 - The `comux` input file generated by `Gurthang`. 79

4.23 Example Bug 3 - The target web server crashing when given the input file. . 79

4.24 Example Bug 3 - The web server's postmortem stacktrace (GDB). 80

4.25 Example Bug 3 - The faulty source code causes an invalid read far from the original location. 81

4.26 A histogram of the number of submissions made across the 55 web servers. . 82

4.27 Bug Fix Example - The crash-inducing `comux` file contains a malformed `Cookie` HTTP header. 83

4.28 Bug Fix Example - Running the web server through GDB shows the location of the crash. 84

4.29 Bug Fix Example - The source code reveals the bug: the failure to check for a `NULL` return value from `strtok_r()`. 85

4.30 Bug Fix Example - Running the same input file with a *newer* version of the same web server shows it no longer crashes. 86

4.31 Bug Fix Example - The newer submission's source code shows the participant added a return value check to fix the bug. 87

Chapter 1

Introduction

Computing has become central to life in the twenty-first century. This is exemplified by the host of services accessible via the Internet. Many of these services are powered by special computer programs called **network servers** (or **web servers**) that communicate with users via the Hypertext Transfer Protocol (HTTP). Often times these servers deal with private and sensitive information that, if compromised, would create serious problems for users. Present risks mean those who use an online service must trust the security of these web servers. The security of a web server, just like any other computer program, directly correlates to the number of exploitable bugs it harbors.

If a bug that was overlooked during a web server's development is discovered by a malicious user on a live web server, it may open the door to denial of service attacks (i.e. bringing down a service and preventing others from accessing it), information theft (stealing user information), or other forms of compromise. Thus, to reduce the risk of catastrophic attacks, the discovery and elimination of as many bugs as possible must happen during development, *before* a web server is used to host a live, Internet-connected service.

Bug discovery during development heavily depends on the testing methods used. One such technique, called **fuzzing** (or **fuzz-testing**), tests a program repeatedly in an automated fashion. Special programs, called **fuzzers**, exist specifically to perform this technique on other applications (**target programs**). Fuzzers operate by repeatedly sending specially-crafted inputs (called **test cases**) to a running instance of the target program. During each

run, the fuzzer taps into a feedback loop to learn *how* the target program processed a test case. Fuzzers create new test cases on the fly by making small incremental adjustments to existing test cases, or by generating new ones from scratch. In either case, many fuzzers employ several strategies to create test cases that are capable of triggering new behavior in the target program. Fuzzers use the knowledge of new execution behavior as a way to guide their efforts toward the discovery of new bugs. A fuzzer's feedback loop depends on its implementation and the manner in which the target program is available. Some fuzzers simply observe the target program's output to discover changes in behavior. Other fuzzers use more advanced techniques to compile a special version of the target's source code capable of informing the fuzzer exactly *which* parts of the target program were executed when processing a test case; this is called coverage-guided fuzzing. What feedback is available to a fuzzer dictates the selection of future test cases.

1.1 Challenges of Fuzzing Network Applications

Web servers and other network-based applications behave differently than typical programs targeted by most fuzzers because the manner in which a web server receives input is different from that of programs which take input from files or input streams. In addition, there are other qualities to web servers and the network stack that are challenging to fuzz. Below, we have outlined three main challenges of fuzzing network applications that our research pursues.

Challenge 1 - Input Management. Network applications communicate with other machines across a network. As such, they read input through bytestreams over active connections with clients, which are not pre-connected at startup time and are typically created on demand. This constraint means that state-of-the-art fuzzers are not compatible with

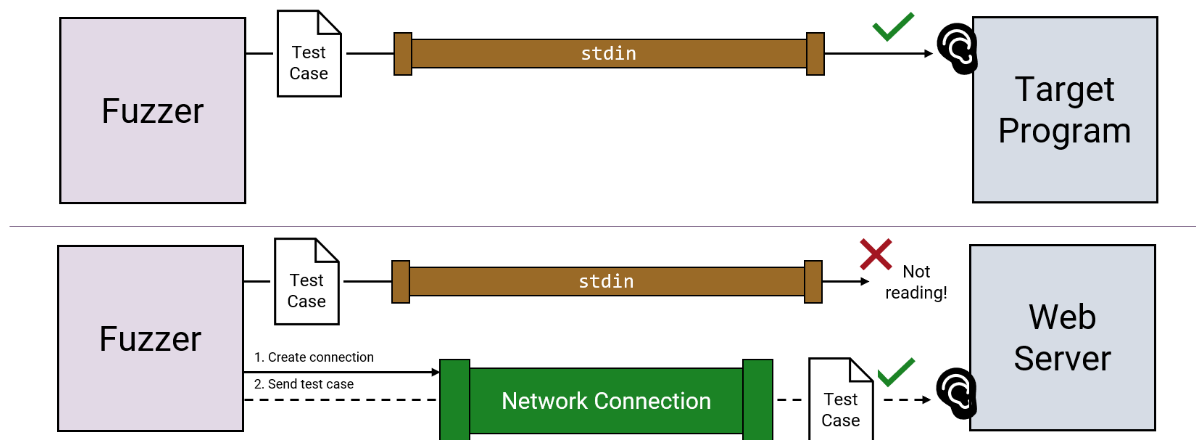

Figure 1.1: **Challenge 1 - Input Management**. The common case (top) allows for fuzzers to make use of the standard input stream to transmit test cases to the target program. Network servers (bottom) can read input only through a dynamic network connection.

network applications and web servers, due to their assumption of a standard bytestream automatically opened at runtime. This makes fuzzing web servers difficult. Web server fuzzing thus requires modification of source code (a tedious task) or a custom solution. Successful attempts have been made to develop libraries that integrate web server fuzzing with state-of-the-art fuzzers, but certain limitations exist in these solutions, such as the inability to handle multi-threaded applications and the need for source code modification [45].

Challenge 2 - Fuzzing Across Multiple Connections in Varying Order. An effective web server must support multiple concurrent client connections. Either by managing multiple threads or using asynchronous event handling, servers multiplex several connected clients at once to prevent one client from hogging the server and delaying the service of other clients. Additionally, the server must gracefully handle any unexpected delays, and must be prepared for unpredictable orderings of connection creation and message delivery. Existing solutions for fuzzing web servers do not support the use of multiple concurrent client connections [39, 45]. This lack of support means that any bugs relating to shared state and concurrency likely will not be found during fuzzing, since only *one* connection is tested for each execution of

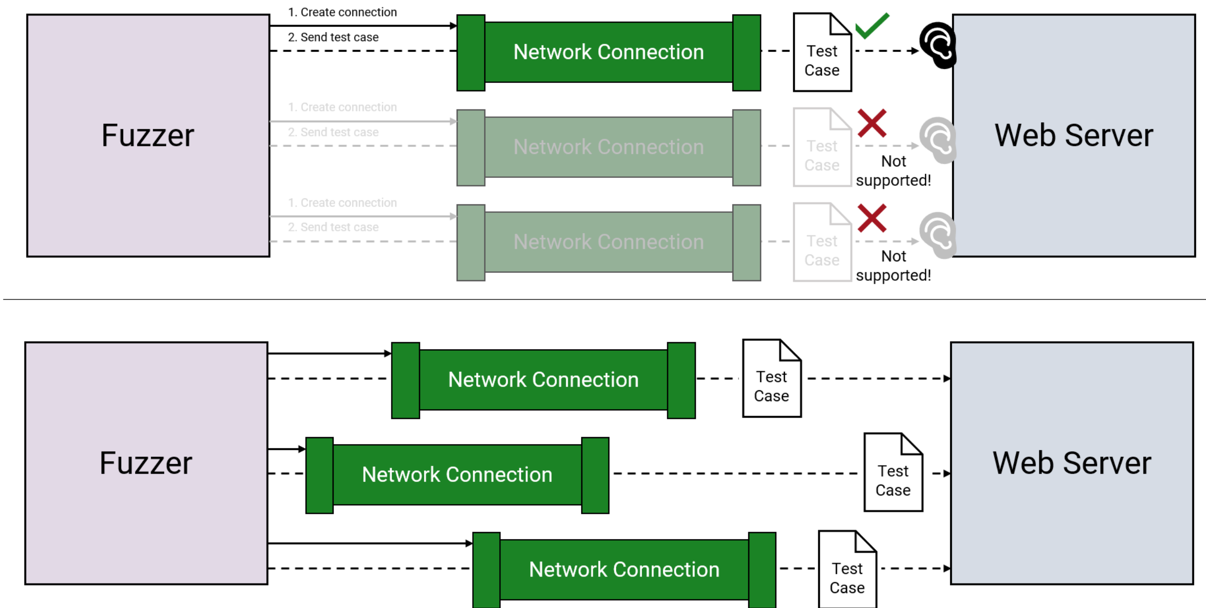

Figure 1.2: **Challenge 2 - Fuzzing Across Multiple Connections of Varying Order**. Existing network server fuzzing solutions can only send a single test case across a single connection to the target web server (top). Not only must multiple connections be exercised to discover concurrency bugs, but the order in which data arrives through these connections should also be changed to examine the server's behavior in the fact of varying message ordering. (bottom).

the target program.

Challenge 3 - Fuzzing Message Boundaries. Network protocols implement several mechanisms for ensuring reliable delivery of data across a connection. In the Transmission Control Protocol (TCP) and Internet Protocol (IP), one application-level message may be split into several packets when sent across a network connection [35]. Underlying constraints in the network stack may affect the manner in which these packets are received by a network server. One example is the size of the receiving machine's kernel packet queue. Another example is the possibility of delays between the delivery of packets. Additionally, the server itself may not provide a buffer with enough memory to read a full message in a single iteration of its logic. These constraints often require network servers to perform multiple

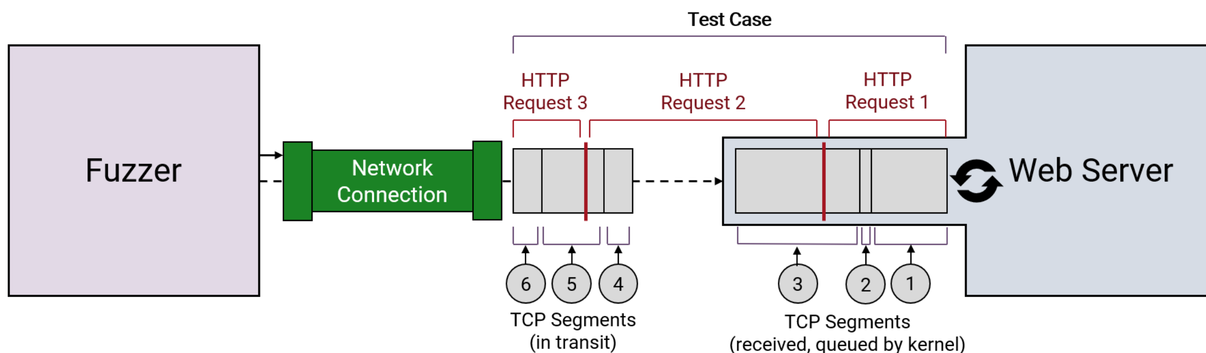

Figure 1.3: **Challenge 3 - Fuzzing Message Boundaries**. This figure depicts three HTTP messages, split among several TCP segments, being sent to a web server. The receiver's queue has buffered segments 1-3, comprising the first HTTP message and a portion of the second message. Segments 4-6 are still in transit.

read operations before seeing a full application-level message. Thus, web servers must be able to recognize the message boundaries of the protocol they are communicating with and choose when to stop listening and start processing. Just as many state-of-the-art fuzzing solutions for web servers do not support multiple concurrent connections, they also lack the ability to manipulate the ordering and timing with which the data is sent across each connection in order to test the server's ability to process message boundaries and handle delays.

1.2 Proposed Solution

To address these challenges we developed `Gurthang`, a framework for web servers that enables fuzzing with much less developer effort and introduces new fuzzing strategies. `Gurthang` takes advantage of coverage-guided fuzzing by instrumenting the target web server's source code using existing state-of-the-art fuzzing tools. `Gurthang` supports the fuzzing of multiple concurrent connections as well as the ordering of data delivery. It can fuzz both single-threaded and multi-threaded servers and does not require the modification of the target

server's source code. Three main components make up `Gurthang`:

- We designed the **Input Bridge** as a library that enables web servers to read fuzzer test cases from the standard input stream without any modification to the source code. We achieved this by implementing a shared library we interpose at run-time. The input bridge establishes internal connections to the target web server and feeds bytes from the fuzzer through to each active connection. Our implementation makes these internal connections indistinguishable to what the web server would interact with in a native environment.
- We designed the **The `Comux` File Format** to specify the content and ordering of multiple connections with the target web server, all from a single file. We organized the information in each `comux` file to allow the other two major components of `Gurthang` to articulate exactly *what* to send to the target server, *which* connection to send it across, and *when* to send it relative to the other connections.
- We designed the **The `Comux` Mutator** to accept existing fuzzer test cases (in the form of `comux` files) and perform mutations to produce new test cases. This component orchestrates all mutations during a fuzzing campaign and is capable of mutating both the data being sent to the target web server *and* the manner in which it is sent.

We created `Gurthang` to address the three challenges described above. `Gurthang` addresses **Challenge 1** with its **Input Bridge**. It feeds fuzzed test cases to the target web server through active network connections. We accomplish this while maintaining the server's native behavior and avoiding the modification of its source code. `Gurthang` addresses **Challenge 2** with both its **Input Bridge** and **`Comux` File Format**. `Comux` files specify a number of connections to be established with the target server, the exact content to send through each connection, and the order in which connections are established and data is sent. The

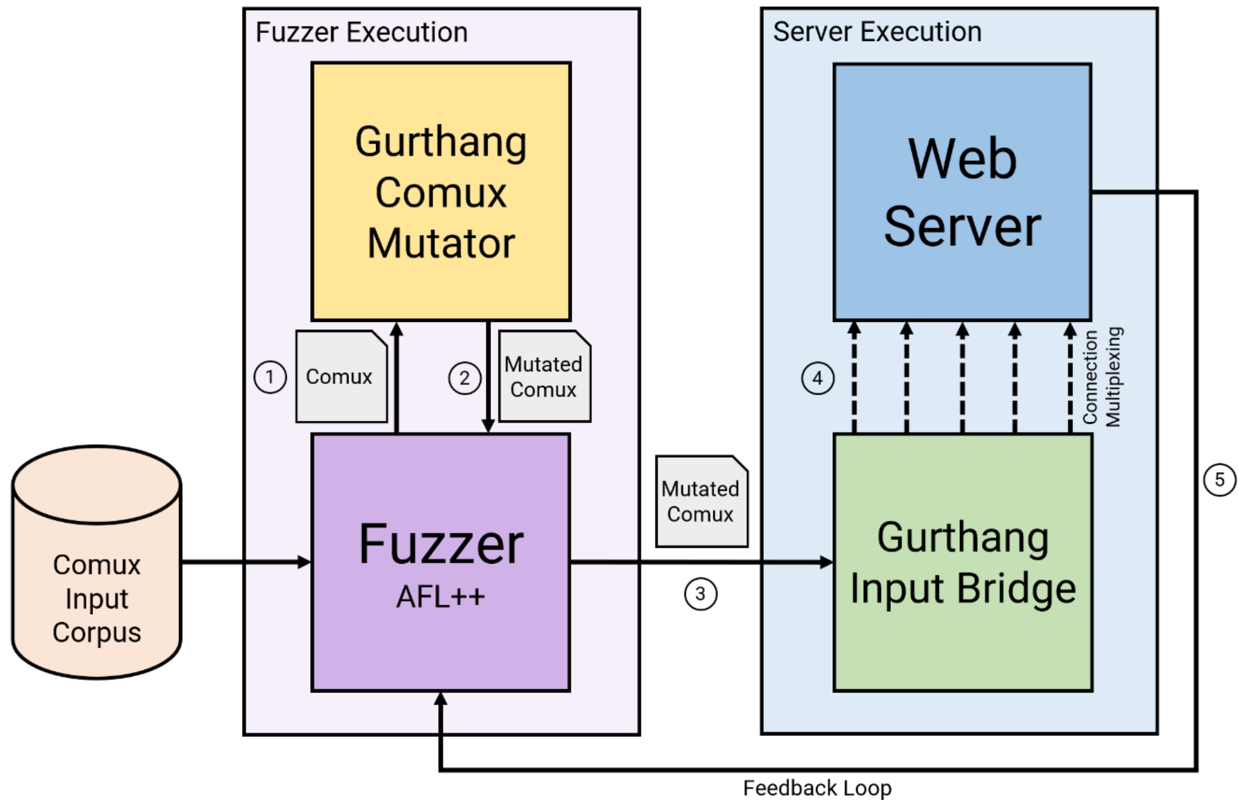

Figure 1.4: A high-level view of `Gurthang`'s architecture.

Input Bridge parses a `comux` file and carries out the creation and management of these connections exactly as specified. Thus, through the contents of a single test case, `Gurthang` can fuzz multiple concurrent connections with the target web server. `Gurthang` addresses **Challenge 3** with its **`Comux` Mutator**. It uses several strategies to create new test cases. Some of these strategies operate directly on the metadata stored within a `comux` file. This metadata specifies the order and manner in which connections are to be established and data is to be sent. Modifying `comux` metadata means the **`Comux` Mutator** can change both the data sent to a web server and vary the instructions that describe *how* the data is to be sent across the network. Our high-level architecture of `Gurthang` is shown in Figure 1.4.

1.3 Contributions Made

Our goals with this research are to simplify the fuzzing process for a variety of web servers, enable the fuzzing of multiple concurrent connections to the same web server, and show that `Gurthang` is capable of discovering bugs in web servers. We make the following contributions:

- **A novel and simple approach to fuzzing web servers**. While varying solutions exist for fuzzing web servers, `Gurthang` is novel in its ability to establish multiple concurrent connections with the target web server in a single execution during a fuzzing campaign. It also introduces simplicity and ease-of-fuzzing; users need not modify the source code of target web servers, and fuzzing can occur simply by loading the `Gurthang` libraries at run-time. The input bridge loads into the web server and the `comux` mutator loads into the fuzzer.

- **Our open-source implementation of `Gurthang`**. We have open-sourced our fully-functional implementation of `Gurthang` on GitHub to encourage its continued use and evaluation in fuzzing. Extensive documentation and our heavily-commented source code is included in the repository, which is available at `https://github.com/cwshugg/gurthang`.

- **Evaluation across several web servers**. `Gurthang` was evaluated through a study at Virginia Tech that invited students of CS 3214: Computer Systems to utilize `Gurthang` to fuzz web servers as a voluntary part of their coursework. We evaluated `Gurthang` with 55 distinct web servers submitted during the study and discovered 48 bugs. In addition to the study, we evaluated `Gurthang` against two industry standard open-source web servers: Apache and Nginx.

- **Real bug discoveries**. During our evaluation study in Virginia Tech's CS 3214

course, Gurthang discovered bugs in 29 of the 55 servers. We analyzed and discovered 48 legitimate bugs that spanned three major categories: the failure to check return values, out-of-bounds memory writes, and out-of-bounds memory reads. Gurthang did not discover any bugs on Apache or Nginx, but the evaluation still provided insight into Gurthang's ability to easily test a variety of servers.

1.4 Thesis Roadmap

Following this introduction, we discuss background information required to understand this research in Chapter 2. We discuss Gurthang's design and implementation details in Chapter 3. We discuss our evaluation of Gurthang in Chapter 4. We discuss related research in Chapter 5 and an outline of future work in Chapter 6. Finally, we bring a conclusion to this research in Chapter 7.

Chapter 2

Background

2.1 Fuzzing

Fuzzing (or **fuzz-testing**) is an automated software testing and security technique that involves giving a program unexpected input with the intention of crashing the program or altering its behavior [15]. Finding an input that crashes the program-under-test (**target program**) opens the door for exploitable vulnerabilities, which will be described in Section 2.1.6. Fuzzing is done in an automated fashion by computer programs (called **fuzzers**) developed for this sole purpose. Fuzzers automate a simple question: which test cases trigger crashes or undefined behavior when a program processes it? They can attempt hundreds or thousands of test cases per second while using this question to search for undiscovered program behavior to find bugs.

2.1.1 The Origins of Fuzzing

The concept of fuzzing was brought into existence by Barton Miller at the University of Wisconsin-Madison in 1989 [15, 36]. Originally, this was in the form of a class project where students wrote programs to generate random inputs and command-line arguments for Unix utilities to seek out flaws in the utilities' code. Miller and other researchers revisited this in 1995 to perform fuzzing on a number of open-source and closed-source Unix and Windows

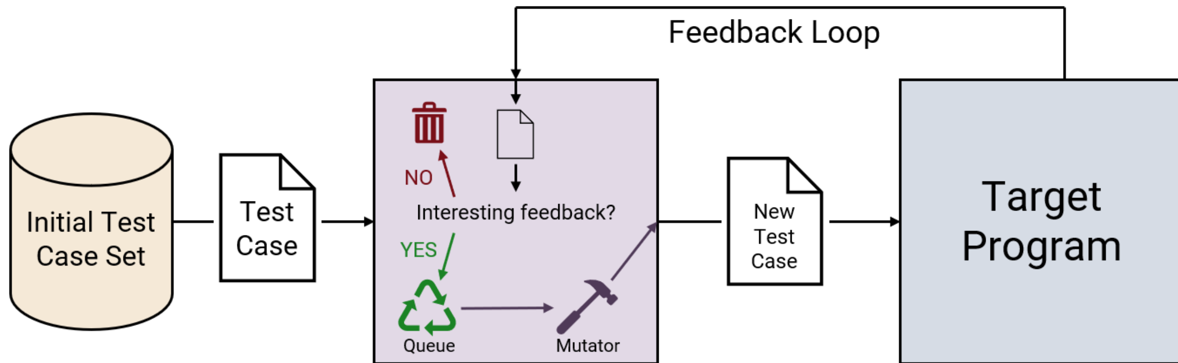

Figure 2.1: Depiction of a standard fuzzing loop. The fuzzer utilizes its feedback loop to decide how to produce future test cases.

utilities [32]. Since its inception in 1989, fuzzing has grown enormously as a field and has taken on many shapes to become an important part of software testing.

2.1.2 Types of Fuzzers

Several large-scale fuzzers have been developed and refined throughout the past two decades to employ varying strategies. These fuzzers differ in terms of coverage awareness and test case creation strategies.

Coverage Awareness

Miller's original concept captured the essence of what is now known as **black-box fuzzing**. Black-box fuzzing operates on the notion that the fuzzer knows *nothing* about the internal workings of the target program, instead making simple random changes to existing test cases [19]. The only feedback a black-box fuzzer has is the target program's output. In many cases, a program may output the same thing despite having changed its internal behavior. In other cases, it may not output anything at all. This lack of feedback means

black-box fuzzers must rely on a versatile set of initial test cases and intelligent test case creation to exercise as much target behavior as possible [19].

Grey-box fuzzing and **white-box fuzzing** came after black-box fuzzing. As the name suggests, white-box fuzzing indicates having all knowledge about the program's internal behavior and using it to craft suitable inputs. Grey-box fuzzing lies on a spectrum between black-box and white-box fuzzing by assuming some knowledge of the target program's internal behavior.

Grey-box fuzzing, the most explored category, was made both popular and accessible by the creation of `AFL`, a fuzzer capable of understanding how the target program's internal behavior changes when its input changes. `AFL` accomplishes this through its compile-time code instrumentation, which we discuss below in Section 2.1.5 [57].

Grey-box and white-box fuzzers possess a feedback loop more capable of understanding the target program's behavior during execution. This allows them to learn which test cases are more successful at triggering new behavior in the target program and use this knowledge to select these cases and derive new test cases from them.

Input Mutation vs. Input Generation

Two main strategies exist for input creation: **mutation** and **generation** [19, 29]. **Mutation** takes a given set of valid inputs (typically called an **input corpus**) and makes small incremental changes to them during the fuzzer's execution. The mutations made typically involve bitwise or bytewise operations, such as:

- Random bitflips
- Inserting bytes

- Removing bytes
- Changing a byte's value

Many general-purpose fuzzers rely solely on test case mutation, as they are not aware of the target program's input syntax and must rely only on an initial set of test cases to mutate in order to achieve results.

Generation takes a specification of the target program's input syntax (or grammar) and generates inputs from scratch that obey this specification. This approach creates test cases that follow the syntax and semantics of the target program's expected input, making it much easier to exercise the target's behavior *past* its initial syntactic and semantic checks [19]. Generation-based fuzzers have been shown to find deeper execution behavior (and bugs) in certain programs [2, 46]. However, this approach is less general and as such syntax-aware generation-based fuzzers require more target-specific development.

2.1.3 Basic Blocks

Many grey-box fuzzers use knowledge of the target program's basic block coverage in their feedback loop for test case creation. The concept of a **basic block** relates to the control-flow structure of a computer program. A basic block is defined as a straight-line sequence of machine instructions containing a single entry point and a single exit point [1]. In terms of assembly code, a basic block would make up the assembly instructions between two conditional branch instructions, because they cannot change course and will always execute in the exact same order. Basic blocks are highly important in compiler design, but have also been used by coverage-guided grey-box fuzzers to measure the control-flow path a target program takes when processing a test case.

2.1.4 LLVM

The LLVM Project (short for **Low Level Virtual Machine**) is a compiler framework that provides advanced compilation infrastructure and a variety of static and dynamic code analysis tools [23]. Some of these tools operate on a program at run-time to detect issues during the program's execution. These are called **Sanitizers**; several exist to fulfill different purposes:

- **UBSan** (**UndefinedBehaviorSanitizer**) detects undefined behavior in a program [50].
- **ASan** (**AddressSanitizer**) detects memory errors in a program [51].
- **TSan** (**ThreadSanitizer**) detects data races in multi-threaded programs [53].

Other LLVM tools enable the tracking of basic block execution throughout a program's execution. LLVM's **SanitizerCoverage** provides compile-time instrumentation that injects calls to custom, user-defined functions at every basic block within a program [52].

2.1.5 `AFL` and `AFL++`

`AFL` stands for **American Fuzzy Lop**. It is a grey-box fuzzer created and originally released in 2014 by Michał Zalewski. It sparked the heavy usage of grey-box fuzzing through source code instrumentation for many fuzzers created thereafter [12, 57]. `AFL` has found bugs across hundreds of open-source applications since its creation, and many researchers have created forks of the original that build on its strengths and pursue specific goals [4, 11, 33, 39, 58].

`AFL` implements a custom compiler pass that instruments the target program with assembly code placed at the start of each basic block [12, 57]. Originally, this instrumented assembly

code was hand-written by Zalewski, but it was later replaced with LLVM's Sanitizer Coverage. At run-time, `AFL` sets up a shared memory region between itself and the target program. When the target executes a basic block, the inserted instrumentation code uses this memory region to record this fact, allowing `AFL` to learn which basic blocks were explored during the target program's execution. When new sequences of basic blocks are detected, `AFL` deems the most recent input as interesting (having triggered new behavior in the target program) and recycles it back into the queue of to-be-tried test cases. In this way, `AFL` achieves a grey-box feedback loop that allows it to reuse interesting inputs based on the target program's behavior.

In the years since `AFL`'s creation, several researchers in the fuzzing community have developed `AFL++`, a fork of `AFL` that vastly improves on `AFL`'s instrumentation abilities and performance [9, 11]. It also adds a higher degree of customizability through custom mutator support, a variety of settings to adjust performance, and more [9].

`AFL++`'s custom mutator support allows for developers to create a shared library that implements an API for `AFL++` to invoke. In this way, `AFL++` can be modified at run-time to become aware of the input syntax of a target program [9]. This feature has been used to create mutation strategies that are aware of the input syntax grammar to make `AFL++` a more effective fuzzer [21]. We chose to use `AFL++`'s custom mutator support in our implementation of `Gurthang`.

Like its predecessor, `AFL++` has been used to find several bugs in many well-known open source projects [9]. Furthermore, it is backed up by a large community of fuzzing researchers who continually make improvements to it.

2.1.6 Memory-Related Vulnerabilities

The OWASP Foundation and the National Vulnerability Database both maintain a running list of known categories of security vulnerabilities found in software [7, 15, 16]. Many of these categories stem from illegal memory operations in type-unsafe languages such as C. They include:

- **Buffer Overflows**. These occur when a program writes to memory past the end of a buffer. It is unknown what was overwritten, and as such often times causes undefined and unpredictable behavior.
- **Null Pointer Dereferences**. These occur when a program fails to check for a `NULL` pointer and attempts to use a pointer variable as if it *was not* `NULL`.
- **Double Memory Frees**. This occurs when dynamically-allocated memory is freed twice. The first time is valid, but the second time will often corrupt the memory allocator's internal data structures, causing a segmentation violation, abort, or other crash.
- **Use-After-Frees**. This occurs when dynamically-allocated memory is accessed *after* it has been freed. Once freed, this memory no longer belongs to the program, and as such a use-after-free can cause undefined behavior, often ending in a crash.
- **Uninitialized Memory Accesses**. As a program executes, portions of its stack and heap memory may be reused. When this occurs, stale, unpredictable values are present in memory. If a program fails to properly reset this memory before accessing it, undefined behavior is likely to occur.

These memory-related vulnerabilities, if left unfixed in a program, may make exploitation of the program possible for a malicious user. With exploitation comes the denial of services for

other users, data theft, and additional situations that can inflict great damage to the software's users. Because of the risks they introduce and how common they are in many software projects, fuzzers are well-suited to finding memory-related vulnerabilities. In addition, they are especially easy to overlook in lower-level programming languages such as C, which has motivated some large software projects, to ban the use of certain unsafe C functions within their source code [55]. While helpful, the banning of unsafe functions does not remove the possibility of creating memory vulnerabilities during development.

2.2 Networking Fundamentals

2.2.1 TCP - Transmission Control Protocol

The Transmission Control Protocol is a network transport protocol that orchestrates the sending and receiving of data across a network between two communicating machines. TCP was originally designed for use by the United States of America's military to ensure reliable delivery of data across unreliable networks (RFC 793) [35]. It is a connection-oriented protocol; before data can be exchanged between two entities, they must first establish a connection by performing TCP's three-way handshake (an exchange of specific indicators to demonstrate a desire to communicate). The entities must also exchange similar acknowledgements before closing the connection at the end of conversation.

TCP is responsible for delivering data from one running process to another, either on the same machine, or across different machines. It utilizes Internet Protocol (IP) addressing to route data from the sender to the receiver. The data is sent across one or more TCP segments, each residing inside an IP packet. TCP builds sequence numbers and other metadata into each segment to detect failures and re-transmit segments when necessary [5]. TCP addi-

tionally uses congestion control mechanisms to protect the network from congestion. When a high number of segments are lost in transit (indicating a congested network), the sender decreases its sending rate and waits longer periods of time for acknowledgements from the receiver (RFC 5681)[25]. On the receiving end, TCP's flow control allows for the receiver to advertise a maximum amount of data it is willing to receive. The data is queued by the receiver's kernel. New data can be queued only when the receiving application empties the queue with a read operation.

TCP's reliability guarantees make it an attractive protocol for network applications. Linux and other operating systems implement TCP within their kernel, providing an abstraction to applications appearing as simple bytestream read/write operations. This abstraction allows applications to use TCP without worrying about reliability. However, applications using TCP may experience delays when networks are congested. They also may experience **short reads**: a situation where less bytes are read from a TCP connection than what was anticipated in a single read operation of the application. Short reads prevent a full application-level message from being received in one iteration in the application's logic. They're caused by delays in TCP segment delivery and the limited size of the underlying kernel's receiving queue. Multiple read operations must be performed to overcome this. Network applications must be implemented to handle the delays and short reads TCP produces.

Most web servers utilize TCP to send messages across a network. These servers thus are implemented to handle delays in data delivery and recognize application-level message boundaries across several read operations. Not only is it beneficial to test a server by manipulating the messages it receives, but it is also beneficial to exercise the server's ability to handle TCP's delays and short reads. Our research seeks to test both aspects through fuzzing.

```
GET /index.html HTTP/1.1
Host: example.server
Accept: text/html
```

Figure 2.2: An HTTP request message depicting a `GET` request for `/index.html`.

2.2.2 HTTP - Hypertext Transfer Protocol

HTTP is an application-level protocol that was designed primarily to organize the transfer of hypertext (formatted text with hyperlinks that makes up web pages) across networks (RFC 7320) [41]. HTTP uses a request/response model. The sender (typically called a **client**) establishes a TCP connection to a recipient (typically called a **server**) connected to the Internet and sends forth an HTTP request message to ask for a resource. The server processes the HTTP request message and sends an HTTP response back to the original sender (the client), which contains the requested resource or some other appropriate error message. HTTP request and response messages are exchanged in a simple, human-readable format. Network applications that communicate with HTTP must conform to this format. If an application violates this message format while communicating across a network, it may cause a lack of service, the premature end of communication, or in extreme cases, undefined behavior and the triggering of bugs in faulty software. Because this research focuses on fuzzing web servers that communicate with HTTP, it is important to understand the rules an application must follow to properly use this protocol, and by extension, the ways in which a fuzzer might be able to intentionally violate these rules to discover bugs.

Request Messages

Request messages are formatted to contain the following lines of text:

```
POST /api/login HTTP/1.1
Host: example.server
Accept-Encoding: identity

{"username": "user", "password": "pass"}
```

Figure 2.3: An HTTP request message depicting a simple authentication request.

1. A request start line
2. A number of HTTP header lines
3. An optional message body

Request Start Line

The request start line consists of three separate pieces, each separated by a single space: a **method token**, the **request target**, and the **HTTP version** [41]. The **method token** is a case-sensitive string specifying what action the requester wishes to perform. Eight different request methods exist as specified by RFC 7231 [42]:

- `GET` requests the current representation of the target resource. (Such as a website's home page.)
- `HEAD` makes the same request as `GET`, but asks that the sender responds only with the response status line and header lines.
- `POST` performs resource-specific processing on the payload send with the HTTP request. (Such as sending authentication information to log into a website.)
- `PUT` replaces all current representations of the target resource with the payload stored in the request message.

- `DELETE` removes all current representations of the target resource from the server's storage.
- `CONNECT` establishes a tunnel to the server by the specified target resource.
- `OPTIONS` asks the server for the usable request methods for the specified target resource.
- `TRACE` performs a message loop-back test along the path to the specified target resource.

The **request target** specifies the particular resource the requester is asking to interact with. Common examples are web pages (`.html` files), style sheets (`.css` files), JavaScript code to run inside a web page (`.js` files), images (`.png` files being one example), and any other file type that might be served as part of a complete web page.

The **HTTP version** specifies the particular version of HTTP the requester is using. By sending this as part of the request, the server will know how to format its own response in order to match the specific HTTP version in use.

Request Header Lines

Both HTTP request and response messages have header lines. Some headers are only used in requests, others are only used in responses, and others are used in both [41]. Request headers are used to attach specific pieces of information to the request. They tell the server *what* resource is being requested and *how* the request should be handled.

Each header consists of two fields: the header *name*, and the header *value*. These two fields are separated by a colon (`":"`). Figure 2.2 depicts a simple HTTP `GET` request with two header lines specifying the `Host` and `Accept-Encoding` headers. Similarly, Figure 2.3 depicts a simple HTTP `POST` request with two header lines specifying the `Host` and `Accept-Encoding` headers.

Request Message Body

```
HTTP/1.1 200 OK
Server: example.server
Content-Type: text/html
Content-Length: 52

<html>
Hello, this is index.html's contents.
</html>
```

Figure 2.4: An HTTP response message depicting a possible response for a `GET` request for `/index.html`

```
HTTP/1.1 403 Forbidden
Server: example.server
Content-Type: application/json
Content-Length: 37

{"message": "Incorrect credentials."}
```

Figure 2.5: An HTTP response message depicting a possible response for a failed login attempt.

HTTP requests can optionally contain a message body. A client can write additional information into the body that may be needed by the server to meet a specific request [41]. Some potential uses of the request message body would be to send the server authentication information or upload a file to the server in a `PUT` request. The message body is placed after all request header fields with a blank line between itself and the final header. Figure 2.3 depicts this with a `POST` request sending a small JSON payload as the request message body.

Response Messages

HTTP response messages are formatted similarly to HTTP request messages, with the only difference being the start line. The start line in an HTTP response message is instead called

the **status line**. Responses still contain a variable number of HTTP headers and an optional message body.

Response Status Line

The HTTP response status line contains three space-separated fields: the **HTTP version**, a numeric **status code**, and a plain text **response phrase**. The HTTP version is formatted exactly the same as the request message's version and indicates the HTTP version the server used to build the response message and process the client's request.

The **status code** and **reason phrase** exist to describe the result of the server's work after processing the client's request. Multiple possible response codes exist for a single request, depending on *how* the client made the request and/or *what* was included in the request. Status codes are three-digit numbers which are grouped into multiple categories [42]. A few common status codes (with their corresponding reason phrases) are listed below.

- `200 OK` - indicates a successful resource access.
- `206 Partial Content` - indicates a successful resource access where only a *portion* of the resource has been sent in the response message.
- `300 Moved Permanently` - indicates that the location of a resource has been moved to a different location. (Used to point the client to the *correct* resource location.)
- `400 Bad Request` - indicates that the client sent a request that was formatted in an unexpected way and that the server cannot or will not process the request.
- `403 Forbidden` - indicates the server understood the client's request but refuses to authorize it.
- `404 Not Found` - indicates the server couldn't find the resources requested by the client.

- `500 Internal Server Error` - indicates the server encountered an unexpected internal error that prevented it from processing the client's request.

Response Header Lines

HTTP response headers are formatted exactly the same as HTTP request headers. Response messages use a different set of headers to send the client pieces of information as to *how* the request was processed and *what* the server is sending back to the client.

Figures 2.4 and 2.5 depict HTTP responses with the `Server`, `Content-Type`, and `Content-Length` headers.

Response Message Body

In the same manner as an HTTP request, HTTP responses can optionally contain a message body to send back additional information to the client. Common uses of the message body in response messages are sending files to the client (such as a `.html` or `.json`) or returning a message to the client in a specific format. Figure 2.4 depicts a response message in which the contents of a requested file (`index.html`) are included in the message body.

Cookies

HTTP has a variety of headers that may be included in a request or response message. One such header is the `Cookie` header. **Cookies** are small pieces of data that clients can receive from a server that persist across multiple request/response exchanges. A client's responsibility upon receiving a cookie is to store it and send it *back* to the server with any subsequent requests. The server, upon receiving this cookie, recognizes it and is able to acknowledge that the client successfully performed some previous request.

The need for cookies arises from HTTP's stateless structure. After a single request/response exchange, nothing about the exchange is typically retained by the server and client. Cookies,

```
POST /api/login HTTP/1.1
Host: example.server
Accept-Encoding: identity

{"username": "correct_name", "password": "correct_pwd"}
```

Figure 2.6: An HTTP request message depicting a simple authentication request.

```
HTTP/1.1 200 OK
Server: example.server
Content-Type: application/json
Content-Length: 32
Set-Cookie: auth_token=eyJhbGciOiJIUzI1N...

{"message": "Login successful."}
```

Figure 2.7: An HTTP response message depicting the server's response to a successful login attempt. Note the `Set-Cookie` header with a shortened example of a cookie.

however, were implemented to add state awareness to HTTP, and thus drive most custom authentication systems on modern web servers (RFC 6265 and 7231) [3, 42].

The `Set-Cookie` header is used by the server in an HTTP response to inform the client of a cookie it should remember. Once the client receives and stores the cookie given in the server's `Set-Cookie` header, it may use it in subsequent requests to the server by placing the cookie in a `Cookie` header within its request message. Figures 2.6, 2.7, and 2.8 depict a series of requests and responses in which a login attempt is successfully made, a cookie is returned to the client, and the cookie is used by the client to request a privileged resource.

Multiple cookies can be specified within the same `Cookie` header by placing a semicolon (`";"`) and a space between each cookie [3]. Web servers must be capable of safely parsing cookies and avoiding undefined behavior when a syntactically-incorrect cookie is sent by a client. The parsing of `Cookie` headers is one part of a web server's implementation that can

```
GET /private.html HTTP/1.1
Host: example.server
Accept: text/html
Cookie: auth_token=eyJhbGciOiJIUzI1N...
```

Figure 2.8: An HTTP request message depicting the client choosing to use a previously-received cookie in order to gain access to a private file.

be directly tested by a fuzzer. Fuzzers can reach this functionality by manipulating `Cookie` headers to remove semicolons, change cookie names, change cookie values, add extraneous characters, and more.

2.2.3 Web Server Concurrency Design

The emergence of Internet-connected services has not only increased the deployment of HTTP web servers, but it has also increased the amount of traffic a web server must handle each day. In addition to this, certain operations performed by a web server may not complete immediately, such as waiting for a connected client to finish sending an entire HTTP request message. These factors require efficient web servers to multiplex several connected clients at once to optimize their performance. To do this, various concurrency models have been created.

The Apache Server Project's **Multi-Processing Modules** (**MPMs**) implements several concurrency models that split the work of accepting client connections and handling client requests into multiple threads or processes [14]. Its `prefork` module implements a multi-process model that splits tasks among the several worker processes spawned throughout the server's lifetime [14]. The `worker` module implements a hybrid concurrency model using multiple processes *and* multiple threads. Similarly, the developers of Nginx (another widely-used HTTP web server) implement a multi-process model similar to Apache's `worker` module. It

spawns multiple worker processes at run-time, each of which uses a threadpool to parallelize read and write operations when communicating with connected clients [18, 48].

These (and other) concurrency models introduce great performance benefit into web servers. This research focuses on multi-threaded web servers; while multiple concurrent threads provide performance benefit, threads often share resources with one another. One example of this is user authentication. If two clients send requests to authenticate in order to access privileged resources, two concurrent threads may need to access the same database (in parallel) to verify the two sets of credentials. To avoid race conditions, these accesses must be protected with thread synchronization, such as Mutual Exclusion locks and/or Condition Variables. Another example is logging. Web servers often write messages to a log for examination by a user. Multiple threads may need to write a message to the server's log in parallel. Again, thread synchronization is needed. When sharing occurs, there is always the possibility of concurrency-related bugs. The ability to test a web server by establishing multiple concurrent client connections may open up the possibility of discovering such bugs.

2.3 Operating Systems Fundamentals

2.3.1 Linux Standard File Streams

In the Linux operating system, **file descriptors** are used by user-space programs to interact with files, networking sockets, pipes, and other abstractions that can be read from or written to. In the C programming language, these file descriptors are represented by simple integers. An API of Linux system calls exist in the C language and can be used by user-space programs to perform read and write operations on the abstractions these file descriptors represent.

When any user-space C program is executed, three file descriptors are implicitly and au-

tomatically opened by the shell for the program to use. These are the **standard input stream**, **standard output stream**, and **standard error stream**.

- The **standard input stream** (`stdin`) has a file descriptor with value `0`. This stream can be used by a program to read input from the user, another process, or a file, depending on how the program was invoked by the user.
- The **standard output stream** (`stdout`) has a file descriptor with value `1`. This can be used by a program to write output to the user, to another process, or to a file, depending on the program's invocation.
- The **standard error stream** (`stderr`) has a file descriptor with value `2`. This stream can also be used for the program to write output to, but it exists to give the program a separate channel to write any diagnostic or error output.

These three built-in file descriptors are commonly used by programs to accept input and display output. Other file descriptors can be opened and closed by the program to perform read/write operations on other system resources. Many fuzzers, such as `AFL` and `AFL++`, utilize the standard file streams (chiefly `stdin` and `stdout`) to feed test cases to a target program. It is a popular choice because of their universal application to many programs, and the fact that shells can attach these file streams to pipes, removing the need for temporary files.

2.3.2 Linux Networking Sockets

As discussed earlier, Linux file descriptors are used to interact with various input/output resources. One such resource is called a **network socket**. Sockets are used by a user-space program to communicate across a loopback network (i.e. on the same machine), a local

network, or even the Internet. A series of Linux system calls create an API for a user-space program to create and interact with sockets in order to communicate with other machines. These system calls are `socket()`, `bind()`, `listen()`, `accept()`, `connect()`, `send()`, `recv()`, and `close()` [20]. Their purposes are described briefly below.

Server-Side Sockets

A web server operates by creating one or more listener sockets and accepting incoming client connections through these sockets. The `socket()` system call is used to allocate a socket file descriptor and return it to the server. Once created, the server selects an open address and port and uses the `bind()` system call to bind the socket to the address and port. `listen()` is then used to register the socket as a *listener* socket; that is, one that will be used to accept incoming connections.

After the socket setup is complete, the server invokes the `accept()` system call to wait for the next incoming client connection. Once a connection is made, a separate file descriptor is created and returned to be used by the server to communicate with the connected client. From here, the `recv()` system call is used to receive bytes sent by the client, and the `send()` system call is used to send bytes to the client.

These system calls provide an abstraction for an application to interact with TCP as if it was a simple bytestream. TCP itself is implemented by the underlying kernel. It is the programmer's responsibility to implement any application-level protocols that sit atop TCP (such as HTTP).

Client-Side Sockets

Clients connect to a listening server. As such, they do not use the `bind()`, `listen()`, and `accept()` system calls. The `socket()` system call is still used to create a new socket file descriptor. However, once created, the client simply invokes the `connect()` system call (with the knowledge of the server's address and port) to establish a new connection to the server.

Once the connection is established, the `send()` and `recv()` system calls are used to send/receive bytes to/from the server. Just like with the server, TCP is abstracted away by this socket programming API, and thus the client's programmer only needs to implement any protocols that sit atop TCP (such as HTTP).

Fuzzing Network Sockets

Network sockets and the Linux standard file streams are both accessed by programs through a file descriptor, but they are treated differently by the underlying operating system. Many state-of-the-art fuzzers can easily support the passing of test cases through standard file streams, but it is much more difficult for them to support this through network sockets. This difficulty stems from internal mechanisms that can restrict the repeated usage of network sockets on a system, such as TCP's congestion control, a Linux system's configuration for loopback connections, and more [25]. This research seeks to address the challenges that come with supporting network sockets to make fuzzing network servers less difficult.

2.3.3 Linux Shared Libraries & `LD_PRELOAD` Interposition

Many programs are written with the ability for users to write modules that can be dynamically loaded to support additional functionality. This is accomplished through a **shared**

library that is loaded into the host program at run-time. Often times, the host program provides developers with an interface of function prototypes that can be implemented to allow for shared libraries to tap into various features of the host. Once implemented and loaded, the host program will invoke the library's functions.

The `LD_PRELOAD` Linux environment variable is used by the Linux loader and allows for a special shared library to interpose new behavior onto a program at run-time. Shared libraries specified in this environment variable are loaded into the host program and utilize the `dlsym()` system call to load existing functions from *other* shared libraries. Because applications make system calls through interfaces provided by shared libraries, an `LD_PRELOAD` library can access these system call interfaces. By defining its own version of a system call's interface, the library can route an application's invocation of the system call through its own definition, while also chaining the call to the original version to maintain the expected behavior. The `LD_PRELOAD` library's own version of a system call interface can force the application to perform additional computation before invoking the original system call. This technique opens the door to modifying a program's behavior at run-time without having to change the program's source code.

An `LD_PRELOAD` library can be used when fuzzing to modify a target program's behavior in order to make it compatible with a fuzzer, thanks to its ability to interpose additional behavior at run-time. This research utilizes this technique to address the challenges of supporting network sockets when fuzzing by interposing behavior onto a variety of networking system calls interfaces.

2.3.4 Linux Process Signals

In the Linux operating system, **signals** are a way for synchronous or asynchronous alerts to be sent to running programs (processes). The delivery of these signals can interrupt the program's execution and force the process to exit, stop running, continue running, run a custom signal handler function, or do nothing at all if the program has chosen to ignore the signal [20]. Several signals exist and are used for different purposes [27]. A subset of them are listed below:

- `SIGINT` - (**Interrupt**) unless handled or ignored, this forces a running program to exit.
- `SIGSTOP` - (**Stop**) unless handled or ignored, this forces a running program to halt execution (but not exit).
- `SIGCONT` - (**Continue**) forces a stopped program to continue execution.
- `SIGSEGV` - (**Segmentation Violation**) this is delivered to a program if an illegal memory access is made.
- `SIGFPE` - (**Floating Point Exception**) this is delivered to a program if an illegal floating point operation is performed (such as dividing by zero).
- `SIGABRT` - (**Abort**) - this is delivered to a program if the `abort()` system call is made. This causes abnormal process termination and is often made when a program detects an unrecoverable error during its own execution.

Signals can be sent and detected with the use of proper system calls in user-space programs, but many signals are generated and sent to a running program by the Linux kernel. Some of these are sent when an illegal operation is performed by the program, indicating a flaw in the program's logic that was potentially overlooked by the developer. `SIGSEGV`, `SIGFPE`, and

`SIGABRT` are three examples of signals that cause the program to abruptly exit when such a flaw is detected (i.e., the program crashes). Fuzzers use signals to detect which test cases cause the target program to crash.

Chapter 3

Design and Implementation

Figure 3.1: `Gurthang`'s logo, created by Connor Shugg.

In this chapter, we describe the design and implementation details of `Gurthang`. We placed the following goals at the heart of `Gurthang`'s design:

- Enable the delivery of fuzzer test cases through socket connections (Section 1.1 challenge 1).
- Enable the fuzzing of web servers across multiple concurrent connections with varying delivery order (Section 1.1 challenge 2).
- Enable the fuzzing of a web server's message boundary processing code (Section 1.1 challenge 3).
- Design a solution that requires no modifications to the server's source code.
- Design a solution that can be used with existing state-of-the-art fuzzers.

3.1 Design Overview

`Gurthang` is a framework to be used in conjunction with existing fuzzers rather than a full-scale fuzzer capable of operating on its own. It is made up of three major components that enable a fuzzer to test web servers. Each of `Gurthang`'s three components serves a different purpose. They are:

- The Input Bridge
- The `Comux` File Format
- The `Comux` Mutator

`Gurthang`'s mutator is invoked by an existing state-of-the-art fuzzer to parse and mutate a given file organized in the `comux` file format. This mutated input is sent by the fuzzer through the standard input stream to the input bridge. The input bridge parses the `comux` file format and establishes connections to the target web server, feeding test case data through each connection. This process is depicted in Figure 3.2.

`Gurthang`'s major components translate directly into separately implemented modules. We implemented the input bridge as a `LD_PRELOAD` library (in C), discussed in Section 3.3. We implemented the `comux` mutator as an `AFL++` custom mutator module (in C), discussed in Section 3.4. We implemented the `comux` file format as an API for the `LD_PRELOAD` library and `AFL++` mutator (also in C), discussed in Section 3.2. This section provides a high-level overview of each component's design while the remainder of the chapter discusses implementation details.

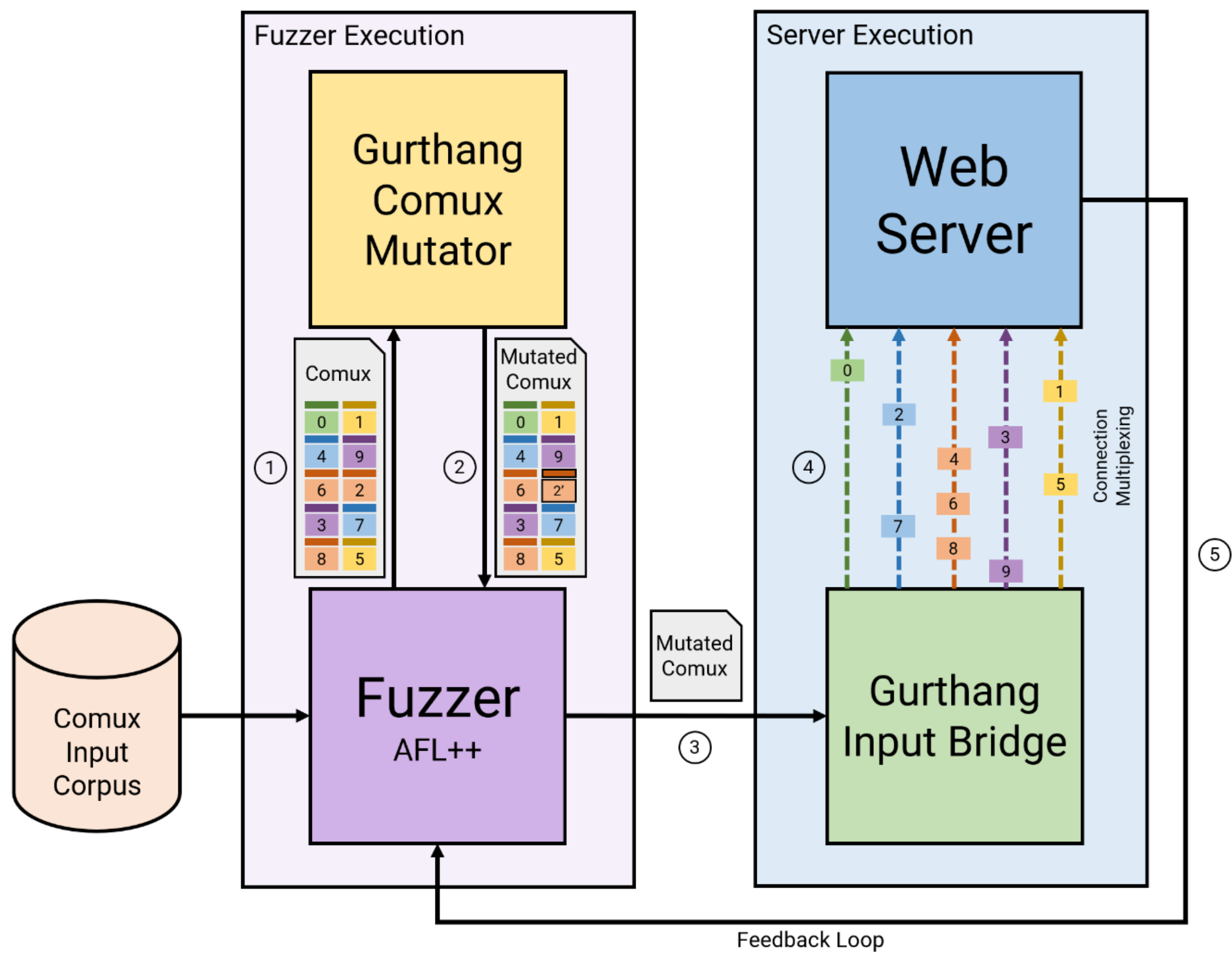

Figure 3.2: A diagram depicting `Gurthang`'s architecture, and its integration into existing fuzzers to deliver mutated payloads across multiple concurrent connections to a web server.

3.1.1 Bridging the Input Gap

To make web servers compatible with some state-of-the-art fuzzers, such as `AFL` and `AFL++`, it is imperative that test cases passed through the standard input stream are fed into a network connection to the target server. We designed `Gurthang`'s input bridge to accomplish this.

The fuzzing framework dynamically loads the input bridge code into the target web server at run-time, which does not require changing a single line of the web server's source code. The bridge captures a copy of the web server's listener socket during its initialization process. Once captured, the input bridge uses this listener socket to create one or more connections to the web server. Once these connections are established, the input bridge awaits input from `stdin` and sends the parsed data through to the web server via the connections, according to the specifications within the `comux` file. Once the server responds, its responses are sent by the input bridge to the standard output stream (`stdout`).

This design injects the expected behavior into to target server at run-time. The connections established by the input bridge allow for the server to behave exactly as it would when communicating with a real client in a native setting; it cannot tell the difference. When reading from `stdin`, the input bridge expects bytes to be formatted in the `comux` file format, which we discuss below in section 3.1.2.

3.1.2 The `Comux` File Format

`AFL` and `AFL++` constrain the contents of a single test case to a single file. In order for `Gurthang` to be compatible with these fuzzers, it also must follow this constraint. However, the input bridge creates *multiple* connections to the target web server and sends *multiple* test cases through these connections. All of this information must be organized within a single file. To address this, we designed the `comux` file format.

The mnemonic `comux` is short for **connection multiplexing**. `Comux` files are made up of chunks. Each chunk contains a payload and is assigned a connection number and scheduling value. The connection number dictates *which* connection the chunk's payload is to be sent through, and the scheduling value dictates *when* the data will be sent to that connection, relative to all other chunks.

`Comux` files allow us to specify the following information:

- How many connections to create with the target server.
- The data to be sent through each connection's socket. (The chunks' payloads.)
- The order in which these connections are established, and the order in which the data is to be sent. (Based on the chunks' scheduling values.)

We designed this file format to contain fields that allow for the specification as to *how* data is sent to the target web server. We designed the third component (the mutator) to perform mutations on the `comux` metadata itself to modify how the input bridge interacts with the target web server during fuzzing. (We will discuss the mutator in Section 3.1.3.) By doing this, `Gurthang` is capable of changing both the data sent to the target server *and* the instructions that specify the manner in which the data is sent by the input bridge.

3.1.3 The `Comux` Mutator

Whereas we designed the input bridge to make web servers compatible with fuzzers, we designed the `comux` mutator to influence the fuzzer itself. An existing fuzzer loads `Gurthang`'s mutator at run-time to make mutations while upholding the integrity of the `comux` file format.

The mutator takes a single `comux` file as input. It parses this file to learn the number

of connections to be made, the number of chunks to be sent, and the order in which to send chunks. Armed with this knowledge, the `comux` mutator randomly chooses a mutation strategy and modifies the `comux` file to produce a new test case for fuzzer. (The format of a `comux` file is not changed during fuzzing.) Through this file format, the `comux` mutator is not only capable of standard `AFL`-like mutations (bitflips, byte changes, etc.) on chunk payloads, it is also capable of splitting chunks, combining chunks, and modifying chunk scheduling values to influence how the input bridge will interact with the target web server. A few examples include:

- Changing the order in which connections are established to the web server.
- Changing the order in which specific chunks of data are sent to the server.

We designed the mutator to uphold the integrity of `comux` files while also changing how `Gurthang` interacts with the target web server.

3.2 The `Comux` File Format

In this section, we describe the implementation details of the `comux` file format.

3.2.1 File Layout

The content of a `comux` file follows the format depicted in Figure 3.3.

The Main Header

The top row represents the **main header**. Its `MAGIC` field is a simple 8-byte pattern used to identify a given file as a `comux` file. We added the `VERSION` field in the event the file

MAGIC	VERSION	NUM_CONNS	NUM_CHUNKS
C1_ID	C1_LEN	C1_SCHED	C1_FLAGS
C1_DATA			
C2_ID	C2_LEN	C2_SCHED	C2_FLAGS
C2_DATA			
...			
C*N*_ID	C*N*_LEN	C*N*_SCHED	C*N*_FLAGS
C*N*_DATA			

Figure 3.3: The Comux file architecture. This diagram orders bytes from left-to-right and top-to-bottom.

format is changed in the future - a 32-bit version number can be specified to support backwards compatibility. NUM_CONNS holds a 32-bit unsigned integer representing the number of connections to be created with the target server. NUM_CHUNKS holds another 32-bit unsigned integer representing the number of data chunks specified in the file.

Data Chunks

Following the main comux header are the N chunks specified by the NUM_CHUNKS field. A chunk represents a collection of bytes that is sent to the target server through a specific connection at a specific time. Each chunk begins with a **chunk header**, which contains the following fields:

- CONN_ID - An unsigned 32-bit integer specifying *which* of the NUM_CONNS connections this chunk's data should be sent to.
- LEN - An unsigned 32-bit integer specifying the number of bytes that make up this chunk's data.
- SCHED - An unsigned 32-bit integer specifying a scheduling value for the chunk. Scheduling values dictate the order in which the chunks in a comux file are sent.

- FLAGS - A 32-bit integer used to represent bit flags to toggle various settings for the chunk.

Immediately following the chunk's header is the chunk's data bytes, whose length is specified by the LEN field. Chunks with lower SCHED values are sent *before* chunks with higher ones, independent of the connection(s) to which these chunks belong. For example: if a comux file contained the chunks as specified in Figure 3.4:

Chunk Number (order in file)	SCHED	CONN_ID
Chunk 0	3	1
Chunk 1	1	0
Chunk 2	0	2
Chunk 3	2	0

Figure 3.4: One example of a possible scheduling of four chunks in a single comux file.

Then data would be sent along three different connections in the following order:

1. Chunk 2's data would be sent first, to connection 2. (SCHED = 0)
2. Chunk 1's data would be sent second, to connection 0. (SCHED = 1)
3. Chunk 3's data would be sent third, to connection 0. (SCHED = 2)
4. Chunk 0's data would be sent fourth, to connection 1. (SCHED = 3)

We chose to include a scheduling field within each data chunk to allow for the sending order of data chunks to be easily modified. This gives the Gurthang mutator (described below) the ability to easily reorder the data sent to a web server, potentially invoking different behavior.

3.2.2 The Comux Toolkit

Along with the `comux` C API, we developed a command-line program to make interaction with and modification of `comux` files easier.

During our evaluation, we used this `comux` command line client program to create an initial input corpus for `AFL++` by converting a number of plain text files containing HTTP requests into `comux` files.

3.3 The Gurthang LD_PRELOAD Library

We implemented `Gurthang`'s input bridge as a dynamic shared library in C in order to override the definitions of a number of system calls typically made by web servers:

- `accept()`
- `accept4()`
- `listen()`
- `epoll_wait()`
- `epoll_ctl()`

By using the `LD_PRELOAD` mechanism discussed in Section 2.3.3 we are able to interpose a program's calls to these system calls.

The `Gurthang` `LD_PRELOAD` library invokes `dlsym()` to look up the addresses of each system call and replaces them with custom implementations defined in the library [28]. The custom version of `listen()` intercepts the web server's listener socket file descriptor and saves a

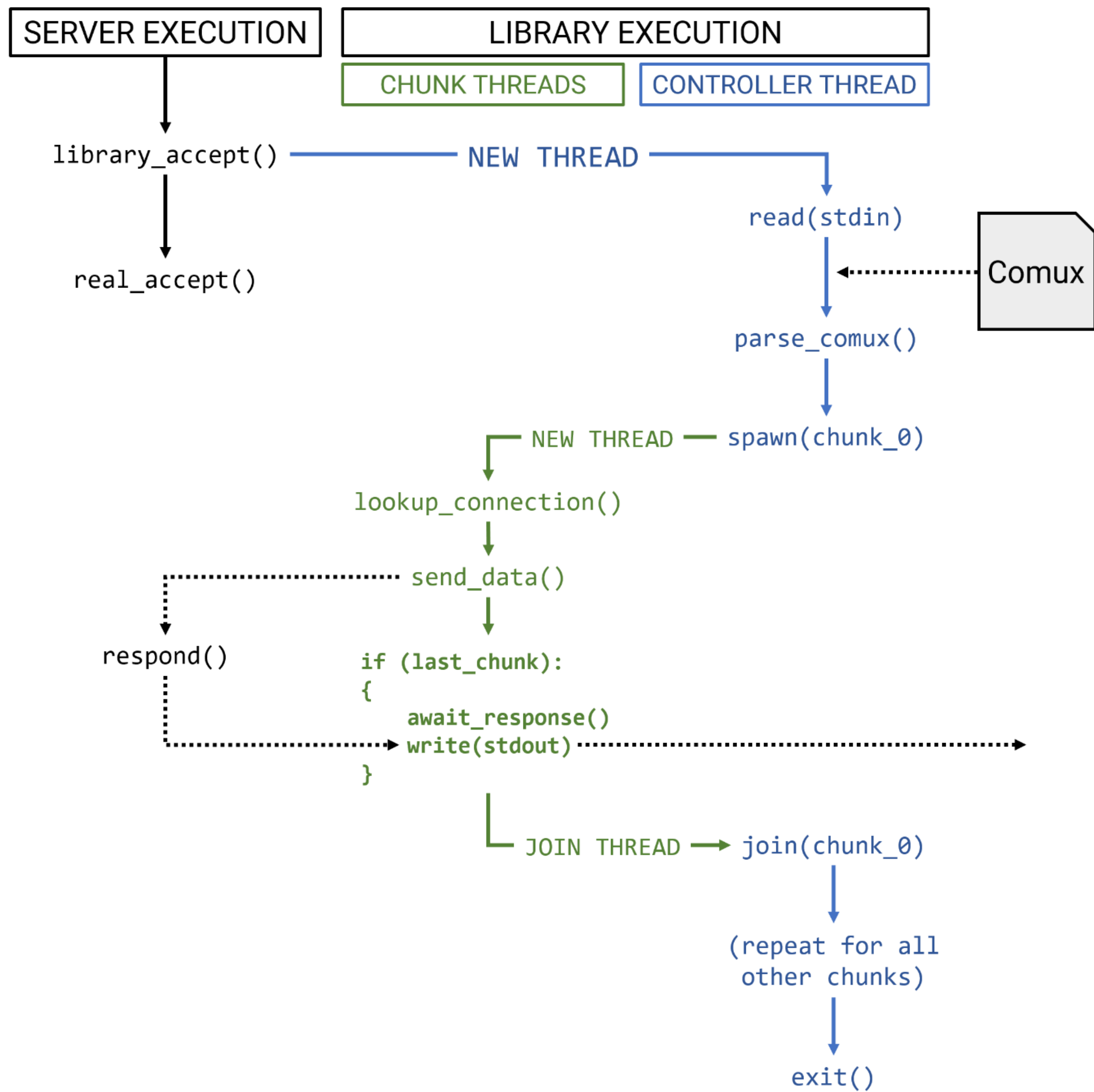

Figure 3.5: A depiction of the internal workings of the Gurthang LD_PRELOAD library, and its interaction with the target web server.

copy. Afterwards, once the server invokes `accept()`, `accept4()`, or an `epoll` function for the first time, the library spawns threads to manage the remainder of the library's work alongside the server's execution.

3.3.1 Internal Threading

The target web server may spawn multiple threads, or it may not. Because of this uncertainty, and because we do not want to hijack server threads to perform the library's work, the `Gurthang` `LD_PRELOAD` library spawns a number of its own POSIX threads [20]. The most important of these is the **controller thread**.

As mentioned above, when the server invokes `accept()` or another appropriate system call indicating the server is ready to accept a new client connection, the `LD_PRELOAD` library's custom version spawns the controller thread. The controller thread initially reads bytes from `stdin` and parses it as a `comux` file. Parsing enables the thread to understand the number of connections that must be made with the target server, as well as the number of `comux` chunks that must be processed.

From here, it manages the spawning and joining of **chunk threads**. Following the order of the chunks' scheduling values, the controller thread spawns a chunk thread for each chunk. The chunk scheduled first is handled by a thread while the controller waits for it to complete. Then, a thread for the second-scheduled chunk is spawned. This repeats until all chunks have been processed, at which the controller thread forces the server (and itself) to exit.

Each chunk thread is responsible for handling the delivery of a single chunk's data to the server across its assigned connection. Once spawned, the chunk thread looks up the correct socket that represents its assigned connection via the connection table (described below). Then, the thread parses the chunk's payload and sends the bytes to the server across the

correct socket. If the controller thread instructed the chunk thread to do so, it will wait for a response from the server after sending its chunk's data. (This will be done if the chunk thread is handling the *final* chunk for a connection.) By spawning a chunk thread for each `comux` chunk, we can easily manage multiple active socket connections by designating a single socket to each thread. Handling all connections in a single thread would require a much more complex design.

3.3.2 Connection Table

Several chunks, despite being spread across the `comux` file, may be assigned to the same connection. The `LD_PRELOAD` library must send all of these same-connection chunks using the same socket file descriptor. On top of this, the order in which chunks are scheduled may interleave communication across multiple connections.

This means each chunk thread needs a way to look up the socket it should use for its assigned connection. We implemented a connection table as part of the `LD_PRELOAD` library to manage all open sockets with the web server. When a chunk thread is spawned, it observes the connection ID number in its `comux` chunk's header. This ID number is used as an index into the connection table to retrieve the connection's status and the socket file descriptor. There are three possible states a chunk thread may find its assigned connection in:

- **The connection has not been opened yet.** In this case, the chunk thread will open a new connection and update the table's entry.
- **The connection is still active.** In this case, the chunk thread simply retrieves the existing socket file descriptor.
- **The connection was closed by the server while handling a previous chunk.**

In this case, the chunk thread exits.

This connection table enables the `LD_PRELOAD` library to support connection multiplexing as defined in the `comux` file format.

3.4 The `Gurthang` `AFL++` Custom Mutator

We implemented `Gurthang`'s `comux` mutator in C as a shared library based on the `AFL++` custom mutator module interface [9]. The `AFL++` custom mutator framework allows for multiple function implementations to let a custom mutator tap into various stages of `AFL++`'s fuzzing procedure. We implemented the `Gurthang` custom mutator to do the following during an `AFL++` fuzzing campaign:

- Inspect `comux` test cases and instruct `AFL++` how many times it should be mutated and executed against the target program.
- Mutate `comux` chunks using several strategies, while maintaining the integrity of the `comux` file format.
- Trim the data within `comux` chunks in order to minimize test case size.

3.4.1 `Comux` Inspection

We implemented `afl_custom_fuzz_count()` in `Gurthang`'s mutator to inspect `comux` test cases and tell `AFL++` the exact number of times a test case should be mutated and executed against the target web server [9]. This gives the `Gurthang` mutator the power to prioritize certain `comux` test cases over others.

We implemented this to favor `comux` files that have multiple connections and/or multiple chunks. With this, the fuzzer will spend more time testing the web server against `comux` test cases specifying a higher number of connections and/or chunks. We chose this with the expectation that `comux` files representing multiple connections or chunks may have a higher chance of revealing concurrency-related bugs in the target.

3.4.2 Comux Mutation Strategies

When `AFL++` passes a test case to `Gurthang`'s mutator, the mutator parses the `comux` information and randomly selects a strategy. These strategies perform a single mutation on one of the chunks stored within the `comux` file.

Because each initial test case we use to seed `AFL++` is in the `comux` file format, the `Gurthang` mutator implements several mutation strategies that are unique to (and depend on) `comux`. It also implements standard `AFL`-like bitwise/bytewise mutations on the data within each `comux` chunk. These strategies are discussed below.

Strategy 1 - CHUNK_DATA_HAVOC

This strategy selects a random `comux` chunk within the input file and performs an `AFL`-like havoc mutation on the chunk's payload. We implemented this simply by invoking the `surgical_havoc_mutate()` function provided by the `AFL++` developers as a helper method for custom mutators [13]. `Gurthang`'s mutator simply provides it with the chunk's data and instructs it to work within the data's bounds.

Strategy 2 - `CHUNK_DATA_EXTRA`

This mutation is similar to the havoc mutation strategy mentioned above. Within this, we implemented a few extra havoc-like mutations we believed to provide interesting test cases. They are:

- **Reverse Bytes** - select a random range of bytes within a `comux` chunk's data and reverse their order.
- **Swap Two Bytes** - select two random bytes within a `comux` chunk's data and swap their positions.

Strategy 3 - `CHUNK_SCHED_BUMP`

We implemented this strategy to intentionally modify the scheduling value of `comux` chunks. By modifying when chunks are scheduled to be sent to the target web server, the `Gurthang` `LD_PRELOAD` library will establish connections and/or send chunks of data to the target server in different orders. This exact purpose motivated us to design the `comux` file format with a scheduling field in each chunk. By reordering the sending order of chunks, the `Gurthang` mutator may increase the chances of detecting new behavior within the target server.

To perform this mutation, the `Gurthang` mutator searches through the `comux` file for a suitable chunk that has enough room to make a difference if its scheduling value is updated. A suitable chunk has the following properties:

1. The chunk must have a scheduling value that, if increased or decreased, will still maintain the same relative ordering within its own connection.
2. The chunk must, by having its scheduling value increased or decreased, be scheduled

differently than it was before, with respect to its neighboring chunks for other connections.

Consider the following example of a simple `comux` file with two connections represented across three chunks:

```
CHUNK       CONN_ID     SCHED
-----------------------------
C-0         0           1
C-1         1           0
C-2         0           2
```

Figure 3.6: An example `comux` file with two connections and three chunks.

The scheduling values dictate that chunk `C-1` will be sent to the web server first. Chunk `C-0` will follow it, and chunk `C-2` will be sent last. Both `C-0` and `C-2` are assigned to connection 0. Because of this, we do *not* want to change their relative order. Why? It is possible `C-2` contains part of a payload that must arrive last along connection 0, after `C-0`'s payload. Changing this order would generate a *completely* different test case that heavily upsets the format of the expected input (such as sending the bottom half of an HTTP message *before* the top half). Mutation-based fuzzing makes small, incremental changes to an existing test case to produce a new one that is *similar* to the original. This similarity is how mutation builds off of previous discoveries. A drastic mutation that heavily modifies an existing test case is not likely to trigger new behavior in the target program. Instead, it would likely trigger already-explored error paths.

In this case, the `CHUNK_SCHED_BUMP` cannot increase `C-0`'s scheduling value up to 3. It could be increased up to 2, but this would not change the order in which the chunks are sent.

However, it is suitable to decrease its scheduling value down to 0:

```
CHUNK       CONN_ID     SCHED
-----------------------------
C-0         0           0
C-1         1           0
C-2         0           2
```

Figure 3.7: The same example `comux` file as shown in Figure 3.6, with a new scheduling value for chunk `C-0`.

The result of this scheduling bump means that chunk `C-0` will now be sent first, followed by `C-1` and `C-2`. The relative ordering between `C-0` and `C-2`, which belong to the same connection, is still maintained. At the same time, this mutation has created additional delay between the sending of `C-0` and `C-2` by establishing connection 1 and sending `C-1` between them. This may uncover more interesting target behavior.

Strategy 4 - `CHUNK_SPLIT`

This mutation selects a random suitable chunk and splits it into two separate chunks. We implemented this strategy with the understanding that splitting a single chunk into *multiple* chunks may open the door for more future `CHUNK_SCHED_BUMP` mutations, allowing payloads to be sent to the target server at different times and in different orders. The mutator selects a suitable chunk in the same way as described above in the `CHUNK_SCHED_BUMP` strategy. There must be enough room for scheduling variance among the selected chunk's same-connection neighbors to split it while maintaining the same delivery order for the corresponding connection.

Strategy 5 - `CHUNK_SPLICE`

This mutation performs the *opposite* action of `CHUNK_SPLIT`. It selects two neighboring same-connection chunks and combines them into one chunk, randomly choosing a new scheduling value. We implemented this strategy simply to negate `CHUNK_SPLIT`, in order to keep test cases from growing too large in chunk count.

Strategy 6 - `CHUNK_DICT_SWAP`

The `Gurthang` mutator supports the use of dictionaries in the form of text files. At run-time, a user can provide one or more dictionaries through an environment variable. Each dictionary is required to contain one word per line. This mutation strategy is enabled only if dictionaries are given by the user. If the mutator finds a keyword within a randomly-chosen dictionary inside a `comux` chunk's payload, it will swap it out for a different, randomly-chosen word in the same dictionary.

We implemented this with HTTP in mind, as certain inherent dictionaries exist in the protocol. Two examples are: the dictionary of strings representing HTTP request methods and a dictionary of HTTP request header names and values (see Section 2.2.2). We implemented this to support more structured inputs and give variety to HTTP test cases. While it allows for new vocabulary to be placed into `comux` chunk payloads, it is still inferior to today's grammar mutators and format-aware fuzzers [2, 21, 46]. Nevertheless, we believe providing such dictionaries can create more variety during a fuzzing campaign.

3.4.3 Test Case Trimming

We implemented a series of trimming functions within the `Gurthang` mutator to cut down test case sizes during fuzzing. When `AFL++` (and our mutator) trims a test case, it does so to decrease test case's size while still ensuring the test case invokes the same behavior in the target program [9].

We implemented test case trimming to uphold the integrity of the `comux` file format by preventing `AFL++` from using its built-in trimming features. `AFL++`'s built-in trimming is not aware of any file format within the input files, and as such `AFL++` would destroy `comux` header information by cutting out random bytes across several trimming steps.

`AFL++`'s trimming procedure works by first choosing a number of trimming steps to perform for one test case. Then, for each trimming step, random bytes are removed, and the trimmed version is executed with the target program. If the same behavior was invoked, the trimming step succeeded. Otherwise, the trimming step failed. Mimicking this, the `Gurthang` mutator's performs the following logic:

1. Parse the `comux` test case and determine the number of chunks present.
2. Choose one chunk at random (chunk `C`).
3. Choose a set number of bytes to remove during each step (`N`). `N` is computed proportional to chunk `C`'s payload size.
4. Choose a set number of trimming steps to attempt (`S`). `S` is the quotient of the `comux` chunk's payload size and `N`.
5. For each of the `S` trimming steps:
    (a) Remove `N` bytes from `C`'s payload.

(b) If trimming succeeded, use the new version of `C` for the following trim step.

(c) If trimming failed, reset `C`'s payload back to its previous state.

(d) If, after 100 trimming steps or 25% of the total trimming steps (whichever comes first), there is a less than 10% success rate, give up on trimming this test case.

Our test case trimming can be summarized as: *keep removing `N` random bytes, one step at a time. If trimming isn't working, quit early.* By removing an amount of bytes proportional to the selected `comux` chunk's payload size, larger chunks take roughly the same number of trimming steps as smaller chunks. Furthermore, `Gurthang`'s mutator observes the success rate over time and will quit trimming if a low success rate is seen, allowing `AFL++` to move onto a new test case faster.

3.5 Design Limitations

We designed `Gurthang` to enable the fuzzing of web servers, and while we have observed our design to work, some of our choices have introduced limitations.

`Gurthang` exercises multiple server threads concurrently by establishing multiple connections through the `LD_PRELOAD` library. Multiple server threads executing concurrently all write to the same shared memory region controlled by `AFL++`. While the `LD_PRELOAD` library's controller thread imposes an order on `Gurthang`'s threads, no such ordering exists for the server's threads. This in addition to the unpredictability of Linux thread scheduling may cause some variance in recorded execution paths of the same test case. These may falsely show up as separate unique discoveries during fuzzing. However, we have observed that this variance does not impede `AFL++`.

`Gurthang`'s input bridge overrides a number of networking system calls to detect when a

web server is ready to accept a new client connection. We override `accept()`, `accept4()`, `epoll_ctl()`, and `epoll_wait()`. It is possible some web servers use other system calls, such as `select()`. Our implementation of `Gurthang` would *not* be capable of fuzzing these servers.

Chapter 4

Evaluation

4.1 Evaluation Goals

Our goals for evaluating `Gurthang` are as follows:

- **Goal 1:** Show `Gurthang` can easily be integrated with `AFL++`, a fuzzer representing the state-of-the-art of source-code-guided fuzzing.
- **Goal 2:** Show `Gurthang` is capable of fuzzing web servers of varying implementation.
- **Goal 3:** Show `Gurthang` is capable of finding bugs in web servers.
- **Goal 4:** Show `Gurthang` is capable of finding bugs that evade detection by existing tools.

4.2 Research Study

We made `Gurthang` available in the Fall 2021 semester to students enrolled in Virginia Tech's Computer Systems course (CS 3214) as a study for this research. We established the study as a voluntary activity where students could utilize `AFL++` (harnessed by `Gurthang`) to fuzz their HTTP web servers, which they developed for their final project in the course. We performed this study not only to measure the effectiveness of `Gurthang`, but also to provide

the students in CS 3214 with a hands-on example of fuzzing and a brief look into the field of Computer Security. Our study was approved by the IRB on November 18th, 2021. The study's IRB protocol number is 21-874. We used `AFL++` version 3.15a for the duration of the study.

4.2.1 HTTP Server Project

We built our study onto the existing web server project in CS 3214. This project required students to implement a variety of features on top of a simple single-threaded HTTP server implementation. Students were provided a repository of code that implemented the following features:

- Accepting connections through an IPv4 address
- Parsing HTTP requests
- Constructing HTTP responses
- Serving static files

Students were required to implement a number of features to improve the provided web server. These features were:

- Accepting connections through both IPv4 and IPv6 addresses
- Handling multiple concurrent clients (implementing multi-threading)
- Servicing both HTTP/1.0 and HTTP/1.1 requests
- Preventing path traversal vulnerabilities

- Providing user authentication support
- Providing API to retrieve a listing of accessible MP4 videos
- Streaming MP4 videos and servicing partial requests via the HTTP `Range` request header

Student implementation were graded for correctness (using a suite of unit tests) and for performance (using a suite of benchmarking tests).

4.2.2 Study Protocol

To participate, the students' first step was to complete the project as normal. As they developed their web servers, they used a series of unit tests to evaluate the implemented functionality of their program. We will discuss these unit tests in Section 4.2.4.

Before further participation, we recommended that students first passed all unit tests to ensure they had a robust web server. (However, the decision was ultimately up to the student.) When ready, participating students would run the fuzzer and consent to having their results and source code collected for research. We acquired informed consent electronically. Students could freely choose to opt out and have any previously collected data deleted.

Consent and interaction with the fuzzer (`AFL++` and `Gurthang`) was done by the participants through a Python script installed on the Computer Science department computing nodes upon which their coursework was done. These 32 computing nodes run CentOS 8 on the following hardware:

- **Machine:** Dell R640 1U server
- **Processors:** 2x Intel Xeon Gold 5218 2.3GHz 16-core processors

- **Memory:** 384 GB @ 2933 MT/s
- **Disk:** 1 TB hard drive
- **Ethernet:** 10 Gigabit interconnect

```
[AFL++ Server Fuzzing Interface]
Usage: fuzz-pserv.py [OPTIONS] --src-dir PATH_TO_SRC
This script can be used to invoke AFL++, an advanced fuzzer, to fuzz your pserv
implementation.
Interested in how it works? Check out the documentation in the project base code.

How to run: at its simplest: fuzz-pserv.py --src-dir PATH_TO_SRC, where PATH_TO_SRC is the
path to your pserv's src/ directory. This script will compile your code and run it with
AFL++. See the command-line switches below to adjust various settings.

[Command-Line Options]
Switches required for fuzzing are marked with a star (*).
   -h  --help                          Shows this help menu
 * -s  --src-dir PATH_TO_SRC           Points the script at your pserv src/ directory
   -o  --out-dir PATH_TO_OUTDIR        Tells the script where to place the fuzzer's output
                                       (default: within your src directory)
   -d  --fuzz-duration SECONDS         Tells the script how long to fuzz your server

                                       (default: 21600 seconds)
   -r  --summary PATH_TO_OUTDIR        Displays a summary of an existing fuzz output
   -v  --verbose                       Prints build output and other verbose messages
   -i  --inspire                       Prints inspirational messages
```

Figure 4.1: The introduction screen displayed by the Python script to the participants.

Upon consenting, the Python script spawned four parallel `AFL++` instances and ran the fuzzer against the participant's web server. The fuzzer ran until one of the following scenarios occurred:

- The fuzzer discovered a bug in the participant's code.
- The fuzzer reached an imposed time limit: 1 hour and 30 minutes.
- The participant chose to stop the fuzzer's execution.

Once the fuzzer exited, our infrastructure reported any discovered crashes to the participant. It also provided students with scripts to assist them with debugging these crashes.

```
[Fuzzing Summary]
The fuzzer ran for 15 second(s).
Output located at: ./fuzz_out_2022-05-18_23-02-30
Found 1 crashes.
   - Of these crashes, 1 were from receiving signal 11.
   - Crash files can be found in the following directories:
        - ./fuzz_out_2022-05-18_23-02-30/fuzz2/crashes
Your server did not hang.

Want to try reproducing a crash or hang? Try this command:
    ./fuzz_out_2022-05-18_23-02-30/fuzz-rerun.sh /path/to/crash/file
For example:
    ./fuzz_out_2022-05-18_23-02-30/fuzz-rerun.sh ./fuzz_out_2022-05-18_23-02-30/
fuzz2/crashes/id:000000,sig:11,src:000335+000120,time:16495,ss_chunk_extra

Want to debug a crash or hang in GDB? Try this command:
    ./fuzz_out_2022-05-18_23-02-30/fuzz-rerun-gdb.sh /path/to/crash/file
For example:
    ./fuzz_out_2022-05-18_23-02-30/fuzz-rerun-gdb.sh ./fuzz_out_2022-05-18_23-02
-30/fuzz2/crashes/id:000000,sig:11,src:000335+000120,time:16495,ss_chunk_extra

Want to see this summary again? Try this command:
    fuzz-pserv.py --summary ./fuzz_out_2022-05-18_23-02-30.
```

Figure 4.2: An example of a crash report displayed by the Python script to the participants after fuzzing concludes.

From here, if they chose, the participant would spend time analyzing the reported crashes and debugging their server with the provided information and scripts. Participants could freely repeat this procedure (fuzzing, then debugging) as many times as they desired.

After each fuzzing campaign, if the participant had consented, our infrastructure sent a copy of their web server source code and fuzzing results to a secure location on the VT Computer Science department computing nodes. Upon granting consent, each participant was assigned a randomly-generated ID string stored within a file in their repository. These participant IDs were used by our infrastructure to associate a participant with his/her submissions. Each

ID was generated randomly and was used to provide anonymity. If a participant chose to repeat the study protocol multiple times, our infrastructure collected multiple versions of their source code and fuzzing results under the same anonymous participant ID. We analyzed this data after the conclusion of the Fall 2021 semester.

4.2.3 Study Participation

At the time of the study, 236 students were enrolled in CS 3214. Upon examination of the collected data, we observed 79 different anonymous participants that had made submissions. However, we did not receive 79 distinct web servers (i.e. one from each participant). CS 3214 students are required to work in pairs on the course projects. We did not take these project partners into consideration, which meant two partners developing the *same* web server could both individually participate in the study. This resulted in similar or identical copies of the same web server being listed under two separate anonymous participant IDs.

We spent time determining which of the anonymous participants had submitted the same server, in order to better quantify the discoveries discussed below. To do this, we utilized the `diff` and `diffstat` Linux utilities to compare each participant's submitted source code with all other participants' source code [26]. For each anonymous participant, we averaged the scores reported by `diffstat` (the summation of total differences) when comparing with all other participants. Then, any other participant whose code produced a `diffstat` score less than *half* of the average was flagged as a potential match. We automated this by creating a shell script. Then, we manually examined each flagged pair of participants to determine if their source code came from the same web server.

We found 12 cases of one distinct web server being submitted under multiple participant IDs. This is depicted in figure 4.3.

Number of Distinct Servers	Number of Times Submitted (under separate participant IDs)
43	1
6	2
4	3
1	5
1	7
Total: 55	

Figure 4.3: Occurrences of distinct servers submitted under separate participant IDs.

From this analysis we determined 55 distinct web servers were submitted across the 79 anonymous participants. We focus on these 55 distinct servers for the remainder of this chapter.

4.2.4 Unit Testing Prior to Participation

As part of the web server project in CS 3214, a series of unit tests are used to grade the functionality implemented by students. These unit tests have been a part of CS 3214 for several semesters (since their creation in the Spring 2018 semester) and have been refined by the teaching staff over time. Some of these tests are denoted as "robustness" tests, and are designed to test the server's ability to handle a number of unexpected inputs. Some examples include:

- Very long HTTP request URIs
- Unexpected `Cookie` header values
- Malformed HTTP request message bodies

These unit tests have been successful at uncovering common issues with students' implementations across several semesters since their creation in the Spring 2018 semester, and are

agreed to exercise all major functionality implemented by the students, save for one recent addition to the requirements: the ability to stream MP4 videos. This new functionality, however, was required for a *separate* assignment within the course, and thus many participants submitted code that lacked the new features. In April of the Spring 2022 semester we improved the unit test suite to exercise the new MP4 streaming features.

Unit Test Coverage

Despite a general agreement of the unit tests' capabilities, we spent time evaluating them to determine how much of students' implemented features were *actually* tested, and by extension, how much room was left for fuzzing to find additional behavior. We did this through code coverage measurements.

In order to measure the coverage of the existing unit tests, we utilized `gcov`, a utility for the GNU C Compiler (GCC) [49]. As discussed in Section 2.1.3, a program's flow of execution can be represented by a graph of basic blocks, linked together by conditional branch instructions. `gcov` is capable of instrumenting a program then measuring the number of basic blocks executed to form a coverage percentage for each compiled C source file. This percentage quantifies exactly how much of the total program behavior was exercised during testing.

Four main C source files are shared by all 55 distinct web servers: `http.c`, `main.c`, `socket.c`, and `bufio.c`. These files originate from the initial code provided to the students at the start of the project. They are also the source files into which most of the students' implementations are written. Because of this, we measured the code coverage percentage for each one across all 55 web servers to quantify the unit tests' coverage.

To measure the coverage for each source file, we compiled every submitted web server with `gcov`'s compiler flags (`-fprofile-arcs` and `-ftest-coverage`) and linker flag (`-lgcov`) to

create instrumented binaries. Following this, we ran the unit tests against the instrumented web server, after which `gcov`'s output was produced and parsed. With the output, we computed average coverage percentages for each source file, for every distinct web server. We combined these averages to form one overall average. These steps are listed below.

1. For each web server:
   (a) For each submission of this web server:
      i. Compile the web server with `gcov` instrumentation.
      ii. Run all unit tests against the web server.
      iii. Record coverage percentages for `http.c`, `main.c`, `socket.c`, and `bufio.c`.
   (b) Compute and record the average coverage percentages for `http.c`, `main.c`, `socket.c`, and `bufio.c` across all submissions of this web server.
2. Compute the overall average coverage percentage for `http.c`, `main.c`, `socket.c`, and `bufio.c` across all web servers.

We calculated the overall averages displayed in Figure 4.4.

C Source File	Percentage of Basic Blocks covered by Unit Tests
`http.c`	74.67%
`main.c`	70.34%
`socket.c`	66.39%
`bufio.c`	78.40%

Figure 4.4: Overall average percentages of the unit tests' basic block coverage among all study participants.

As a whole, this shows that the unit tests exercise the majority of the students' implemented behavior in the four main C source files. `http.c` is of special interest, as it typically contained most of the features implemented by students throughout the course of the project. Each of the four source files have over half of their code covered, with three being close to 75%. The coverage fell short of 100% for multiple reasons:

- As we mention in Section 4.2.4, a subset of the functionality - MP4 video streaming - is a recent addition. No unit tests had been written yet to cover this at the time of the study, even though some participants submitted web servers with these new features. As such, some basic blocks *not* reported by `gcov` belong to this functionality.
- The provided base code performs lots of error checking. This is especially evident in `socket.c`, where several network programming system calls are made to set up the server's communication sockets. Error checks exist for *all* of these invoked system calls and thus make up a considerable portion of the source file's basic blocks. Error checking is good programming practice, but it is very rare that such system calls will fail. The unit tests do not exercise these error checks because of how trivial they are. Code implemented by students also contains similar error checks in many cases. Because these error checks are not likely to occur unless in extreme cases, the unit tests do not exercise them.

Bearing these considerations combined with the coverage measurements in mind, we have shown the existing CS 3214 unit tests are effective, but not perfect. It is very difficult to design a set of unit tests sufficient for testing *every* possible case of malicious input. Unit testing and fuzzing both seek to exercise a large coverage of the target source code, but they exist on a spectrum. On one end of the spectrum there is *too little* testing. This might involve one or two trivial test cases that exercise very little of the target program's behavior.

On the other end of the spectrum is the idea of turning *every* new test case generated by the fuzzer into a unit test. While this second option certainly exercises the most coverage of the target program, it is not practical. Bearing these thoughts in mind, we hypothesized that while the existing CS 3214 unit tests do well to test the major functionality of a student's web server (plus a number of malicious-input cases), fuzzing may be able to uncover new, untested behavior.

Unit Test Results Among Participants

We highly recommended that students who wished to participate in the fuzzing study should first ensure their server passed all unit tests, with an emphasis on the robustness tests mentioned above. 45 of the 55 distinct web servers passed all unit tests in *at least* one submitted version of their code. As such, these 45 servers ensured their web servers exhibited a baseline ability to handle both expected and unexpected inputs gracefully. We categorized these servers as robust. The remaining 10 web servers failed at least one functionality unit test *and* one robustness unit test in all submitted versions of their code. We categorized these servers as *not* robust. These statistics are illustrated in Figure 4.5. Despite many of the servers possessing this baseline robustness, we still observed several bug discoveries by `Gurthang`.

In Section 4.2.5, we discuss the bugs originating from the 55 distinct web servers submitted throughout the course of the study. In Section 4.2.6, we walk through the details of a number of selected bugs that were discovered in these servers.

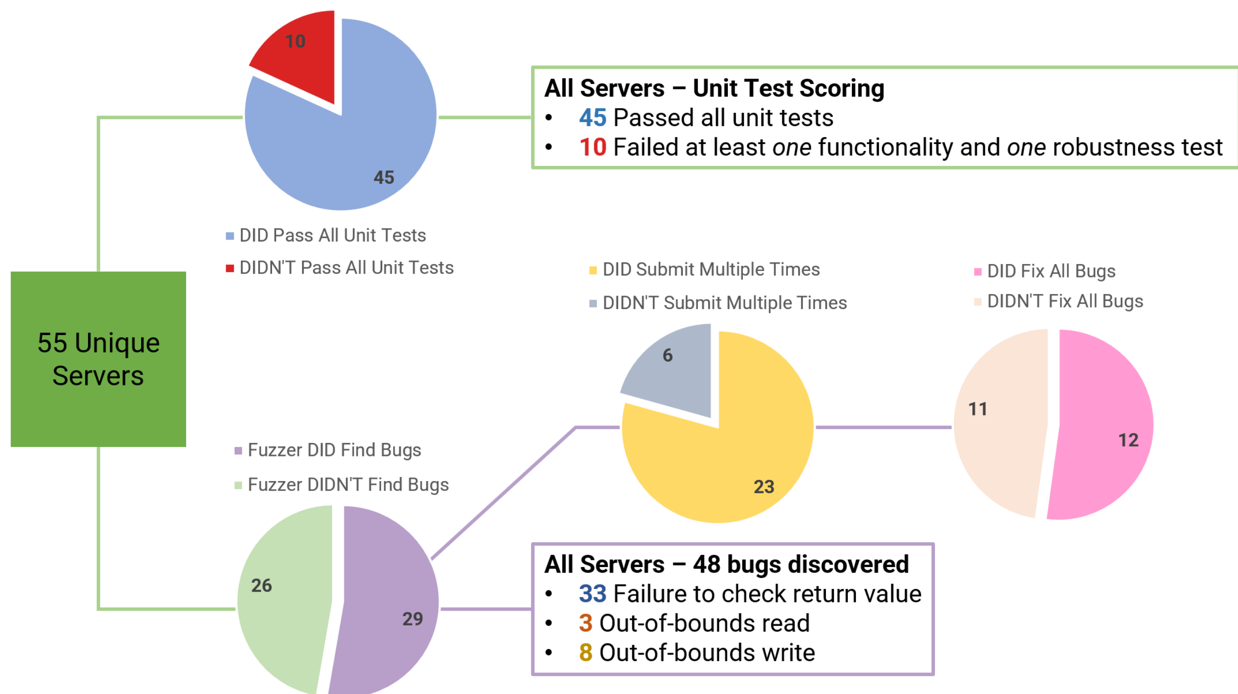

Figure 4.5: A summary of the 55 distinct web servers submitted throughout the CS 3214 study. This depicts unit test scores, discovered bugs, and servers that were submitted multiple times as the participant fixed bugs.

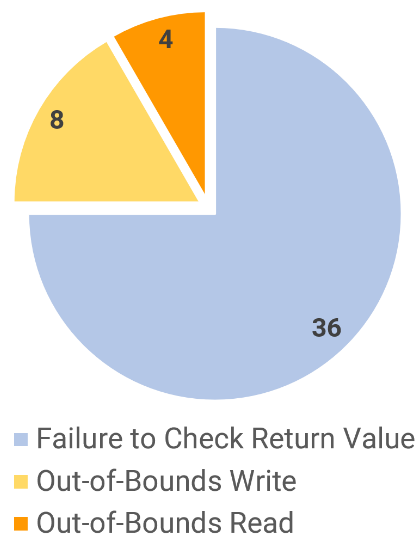

Figure 4.6: The three bug categories determined by examining the 48 discovered bugs.

4.2.5 Assessment of Gurthang's Use By Participants

While every web server originated from the same set of starter code provided as part of CS 3214, each student was required to implement functionality that parsed HTTP requests, formed HTTP responses, parsed HTTP `Cookie` headers, dissected JSON strings, and more. We hypothesized that the implementation of these features opened the door for bug discovery by the fuzzer.

Our results show that `Gurthang` was successful at discovering bugs. Of the 55 distinct web servers, `Gurthang` discovered bugs in 29 of them. 4 of the 29 buggy servers fell into the non-robust category (meaning 4 of the buggy servers failed at least one functionality unit test *and* one robustness unit test). However, the bugs discovered on these 4 non-robust servers were of the same caliber as the 25 robust servers. Because of this, we have combined the two groups' bug discoveries into one discussion. In total, we found 48 bugs across the 55 web servers. We grouped the discovered bugs into three major categories: **failure to check return values**, **out-of-bounds memory writes**, and **out-of-bounds memory reads**. This is depicted in Figure 4.6.

Failure to Check Return Values

The vast majority of bugs we discovered were the result of the programmer's failure to check the return value of various C functions. 36 of the 48 discovered bugs fell into this category. Many of these functions dealt with parsing strings: `strtok_r()`, `strtok()`, `strstr()`, `strchr()`, `opendir()` were all involved, as well as functions in Jansson, a C library dealing with JSON object management [24]. These return values were `NULL` and were passed, unchecked, into other similar functions, including some defined within libjwt, another C library that provides an API for encoding and decoding JSON Web Tokens for the authen-

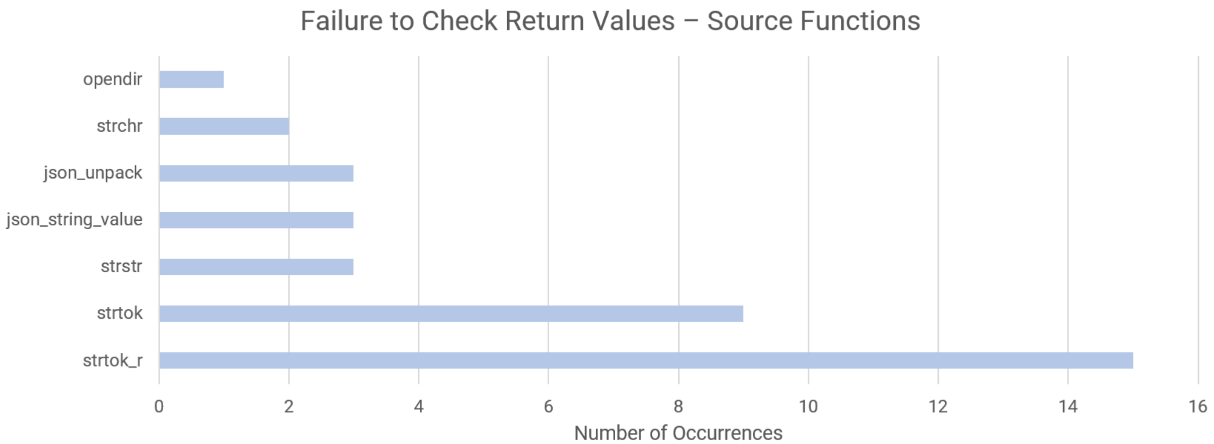

Figure 4.8: Source functions from which NULL return values originated. Some of these functions (json_unpack() and json_string_value()) originate from open-source libraries [24].

tication portion of the project [6]. These return values were also manually dereferenced in some cases. In all cases, the program crashed with a segmentation fault (SIGSEGV). Examples of these bugs are depicted in Figure 4.7.

```c
// ---------- BUG EXAMPLE 1 ---------- //
char* ptr = parsing_function(); // returns NULL
strcmp(ptr, "/something");      // NULL parameter triggers a SIGSEGV.

// ---------- BUG EXAMPLE 2 ---------- //
char* ptr = parsing_function(); // returns NULL
char c = *ptr;                  // NULL dereference triggers a SIGSEGV.
```

Figure 4.7: C code depicting bugs originating from the failure to check a return value.

We depict a detailed breakdown of the source functions from which the NULL return values originated in Figure 4.8.

Out-of-Bounds Memory Writes

The second-largest category of our discovered bugs involved invalid memory writes that modified bytes outside the expected range of a buffer. 8 of the 48 discovered bugs fell into this category.

2 of these out-of-bounds writes were caused by **buffer overflows**, a common software bug involving an attempt to write to memory past the end of a buffer [16]. One of the buffer overflows occurred when 6400 bytes were written into a buffer of a hard-coded size of 1024. Thousands of bytes past the end of the buffer were modified, eventually resulting in an illegal access and segmentation fault. The other buffer overflow was caused by an incorrect `Content-Length` header in an HTTP request sent during fuzzing. A heap-allocated buffer of insufficient size had excess bytes written to it, causing the clobbering of the libc memory allocator's boundary tags. A brief depiction of this type of bug is shown in Figure 4.9.

```
// ---------- BUG EXAMPLE 1 ---------- //
char buffer[1024];
for (int i = 0; i < parsed_length; i++) // 'parsed_length' is 6400
{ buffer[i] = some_value; }
// This loop blindly iterates 'parsed_length' times, which
// in this case is much larger than the hard-coded buffer
// length of 1024.
```

Figure 4.9: C code depicting a buffer overflow bug.

The remaining 6 out-of-bounds writes were caused by **buffer underflows**, a similar bug involving an attempt to write to memory located *before* the beginning of a buffer. All 6 of these bugs were caused by the server parsing and using a *negative* `Content-Length` HTTP header value (the server failed to sanitize the input from the client). Stack-local buffers were initialized with these negative lengths, causing an underwrite (followed by a segmentation

fault) once the buffer was written to. A brief depiction of this bug is shown in Figure 4.10.

```
// ---------- BUG EXAMPLE 2 ---------- //
char buffer[length]; // 'length' is negative
snprintf(buffer, length, "SOME_STRING");
// Memory was written to *before* the beginning of 'buffer',
// overwriting something else nearby on the stack.
```

Figure 4.10: C code depicting a buffer underflow bug.

Out-of-Bounds Memory Reads

Our smallest bug category consisted of out-of-bounds memory reads. 4 of the 48 bugs were caused by the programmer's failure to protect against reading memory outside the bounds of a buffer.

2 of the 4 bugs discovered involved a loop that read memory byte-by-byte until a specific value was found. The fuzzer discovered these bugs by sending an HTTP message that did *not* contain the expected value, causing the loop to read thousands of bytes past the end of a buffer, triggering a segmentation fault (`SIGSEGV`). An example of this is shown in Figure 4.11.

```
// ---------- BUG EXAMPLE 1 ---------- //
char* data = parsing_function(); // returns "STRING NOT CONTAINING DESIRED BYTE"
while (*data != '=')
{ data++; }
```

Figure 4.11: C code depicting an out-of-bounds memory read bug, via an infinite loop.

The remaining 2 bugs originated from the failure to check a buffer's true length, and the assumption its length exceeded some minimum. Attempting to read past the end of the buffer triggered a segmentation fault (`SIGSEGV`). An example of this is shown in Figure 4.12.

```c
// ---------- BUG EXAMPLE 2 ---------- //
size_t length = parse_length(); // length is parsed as 1 byte
char* buffer = malloc(length);  // one byte is allocated
strncpy(dest, buffer, 20);      // assumes length is 20+ bytes
```

Figure 4.12: C code depicting an out-of-bounds memory read bug, triggered by an assumption of a buffer's length.

4.2.6 Examples of Bugs Discovered by `Gurthang`

In this section, we review the details of a number of bugs discovered by `Gurthang` in student web servers submitted as part of the study. Each of the crash-inducing test cases were generated by `Gurthang`'s custom mutator, which allowed `AFL++` to discover the crash. After this, the Python scripts discussed in Section 4.2.2 alerted the participant of the crash that was discovered and provided a shell script to easily debug the crash (shown in Figure 4.1). We walk through the debugging process a student of CS 3214 might take to discover the root cause of each of these bugs. The act of debugging is made easier with the provided Python and shell scripts, but it is still largely the participant's responsibility to use their debugging skills to uncover the root cause of each bug. `Gurthang` contributes the *discovery* of the bug to the participant.

Bug 1 - Failure to Check Return Value - `NULL` Pointer

The first bug falls into the first major bug category discussed in Section 4.2.5: **failure to check return values**. In particular, this bug originates from the participant's failure to check for a `NULL` return value from an unintended call to `strtok_r()` while parsing a malformed HTTP `Cookie` header.

To begin debugging, the participant first examines the contents of the crash-inducing `comux`

file with the provided `comux` toolkit program. Using this, the participant sees that a *single* connection is represented within the file across a *single* `comux` chunk. Inside the chunk is an HTTP `GET` request targeting `/private/secure.html`. A `Cookie` header is specified, but it is malformed; extra white-space surrounds the equals sign (`"="`) in the second cookie. This is shown in Figure 4.13.

```
* COMUX [version: 0] [num_connections: 1] [num_chunks: 1]
* CHUNK 0: conn_id=0, data_length=247, scheduling=0, flags=0x1
GET /private/secure.html HTTP/1.1
Host:elinden.rlogin:27363
User-Agent: python-requests/2.20.0
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-rlive
Cookie: a_bad_cookie1=chocolate_chip; a_bad_cookie = atm al_raisin
```

Figure 4.13: Example Bug 1 - The `comux` input file generated by `Gurthang`. `Gurthang` mutated the second field within the `Cookie` header to contain extraneous white-space surrounding the equals sign (`"="`).

Using the provided scripts, the participant re-runs the server (using `Gurthang`'s `LD_PRELOAD` library) with the crash-inducing `comux` file. Doing this confirms that a segmentation fault is delivered to the web server after it processes the input file, causing the crash. This is depicted in Figure 4.14.

```
$ ../fuzz/fuzz-rerun.sh ../fuzz/fuzz0/crashes/id:000000*
Using port 13650
Waiting for client...
Accepted connection from ::1:49866
Waiting for client...
Header: Host: elinden.rlogin:27363
Header: User-Agent: python-requests/2.20.0
Header: Accept-Encoding: gzip, deflate
Header: Accept: */*
Header: Connection: keep-rlive
Header: Cookie: a_bad_cookie1=chocolate_chip; a_bad_cookie = atm al_raisin
: ader:
Segmentation fault (core dumped)
```

Figure 4.14: Example Bug 1 - The target web server crashes when given the input file.

Next, the participant launches the GNU Debugger to investigate further [10]. It is from here the developer must spend time debugging with his/her existing skills. To be thorough, we describe one possible set of steps a participant might take to debug `Gurthang`'s crash-inducing input file below.

The participant first runs the program and examines the stacktrace upon delivery of the `SIGSEGV` signal. The crash occurred within the web server's routine for handling requests to the authentication-protected directory, `/private`. More specifically, at line 724 in `http.c`. This is shown in Figure 4.15.

```
$ ../fuzz/fuzz-rerun-gdb.sh ../fuzz/fuzz0/crashes/id:000000*
Reading symbols from /opt/scratch/cwshugg/fall2021/research_data/data/0764d01fdbaf
156e09f/sub_1639084819/src/server...done.
fuzz-rerun-gdb.sh: Arguments are set. Type 'run' to reproduce the crash/hang.
(gdb) run
Starting program: /opt/scratch/cwshugg/fall2021/research_data/data/0764d01fdbafeb8
e09f/sub_1639084819/src/server -p 46131 -R /opt/scratch/cwshugg/fall2021/research_
9063094ebcae9fcc81bdaf156e09f/sub_1639084819/src/fuzz_root  < ../fuzz/fuzz0/crashe
0724,ss_chunk_havoc
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
Using port 46131
Waiting for client...
[New Thread 0x7ffff6624700 (LWP 129831)]
[New Thread 0x7ffff5e23700 (LWP 129832)]
Accepted connection from ::1:34034
[New Thread 0x7ffff5622700 (LWP 129833)]
Waiting for client...
Header: Host: elinden.rlogin:27363
Header: User-Agent: python-requests/2.20.0
Header: Accept-Encoding: gzip, deflate
Header: Accept: */*
Header: Connection: keep-rlive
Header: Cookie: a_bad_cookie1=chocolate_chip; a_bad_cookie = atm al_raisin
: ader:

Thread 4 "server" received signal SIGSEGV, Segmentation fault.
[Switching to Thread 0x7ffff5622700 (LWP 129833)]
0x00007ffff6b9818e in __strcmp_avx2 () from /lib64/libc.so.6
Missing separate debuginfos, use: yum debuginfo-install openssl-libs-1.1.1k-6.el8.
(gdb) backtrace
#0  0x00007ffff6b9818e in __strcmp_avx2 () from /lib64/libc.so.6
#1  0x0000000000402847 in handle_private (ta=0x7ffff5621d90, basedir=0x7ffffffd65
a/data/0764d01fdbafeb8bdf367c42e8285476f069063094ebcae9fcc81bdaf156e09f/sub_163908
#2  http_handle_transaction (self=self@entry=0x7ffff5621ee0) at http.c:724
#3  0x0000000000401cf4 in work_it (arg=0x606cc0) at main.c:47
#4  0x00007ffff6e9e1ca in start_thread () from /lib64/libpthread.so.0
#5  0x00007ffff6b09d83 in clone () from /lib64/libc.so.6
```

Figure 4.15: Example Bug 1 - The web server's post mortem stacktrace (GDB).

The participant then sets an appropriate breakpoint and discovers a `NULL` return value from the `strtok_r()` function that is passed into a call to `strcmp()` unchecked. This discovery is depicted in figure 4.16. The failure to check `strtok_r()`'s return value is the root cause

of this bug.

```
Thread 4 "server" hit Breakpoint 1, handle_private (ta=0x7ffff5621d90, basedir
search_data/data/0764d01fdbafeb8bdf367c42e8285476f069063094ebcae9fcc81bdaf156e
648             if (strcmp(clientCookieName, COOKIE_NAME) || jwt_decode(&clien
JWT_CODE, strlen(MY_JWT_CODE))) {
(gdb) list
643         while (currCookie != NULL) {
644             char* endptr2;
645             char* clientCookieName = strtok_r(currCookie, "=", &endptr2);
646             char* clientCookie = endptr2;
647             // check for correct cookie name and JWT
648             if (strcmp(clientCookieName, COOKIE_NAME) || jwt_decode(&clien
JWT_CODE, strlen(MY_JWT_CODE))) {
649                 currCookie = strtok_r(NULL, "; ", &endptr);
650                 continue;
651             }
652             // check for expired token
(gdb) print clientCookieName
$3 = 0x0
```

Figure 4.16: Example Bug 1 - Examining the `NULL` pointer returned from a call to `strtok_r()`.

Bug 2 - Out-of-Bounds Write - Buffer Underflow

The second bug forces the target web server to overwrite an important stack-local value and cause a segmentation fault. It falls under the second major bug category: **out-of-bounds writes**. With the provided `comux` toolkit, the participant may first view the contents of the crash-inducing `comux` file. Doing this shows that the `comux` file represents a *single* connection across a *single* `comux` chunk. The chunk contains a simple HTTP `POST` request to the `/api/login` endpoint, supplying a JSON request body with login credentials. `Gurthang` mutated this file to have a *negative* value for the HTTP `Content-Length` header. This is depicted in Figure 4.17.

```
* COMUX [version: 0] [num_connections: 1] [num_chunks: 1]
* CHUNK 0: conn_id=0, data_length=152, scheduling=0, flags=0x1
POST /api/login HTTP/1.1
Content-Length:-47
Accept-Encoding: identity
Host: hornbeam.rlogin:29826

{"username": "user0", "password": "RVIfLBcbfn"}
```

Figure 4.17: Example Bug 2 - The `comux` input file generated by `Gurthang`.

Using the provided scripts, the participant re-runs the server with the `comux` file to confirm the crash. This action is shown in Figure 4.18.

```
$ ../fuzz/fuzz-rerun.sh ../fuzz/fuzz2/crashes/id:000000*
Using port 13650
Accepted connection from ::1:49964
Segmentation fault (core dumped)
```

Figure 4.18: Example Bug 2 - The target web server crashes when given the input file.

After confirming the crash, the participant launches the GNU Debugger to investigate [10]. The stacktrace at the time of the segmentation fault shows the crash occurred within a call to libc's `realloc()` function when it was given an inaccessible memory address. This is shown in Figure 4.19.

```
$ ../fuzz/fuzz-rerun-gdb.sh ../fuzz/fuzz2/crashes/id:000000*
Reading symbols from /opt/scratch/cwshugg/fall2021/research_data/data/1efe605e
b1cee05/sub_1638926482/src/server...done.
fuzz-rerun-gdb.sh: Arguments are set. Type 'run' to reproduce the crash/hang.
(gdb) run
Starting program: /opt/scratch/cwshugg/fall2021/research_data/data/1efe605e34d
ee05/sub_1638926482/src/server -p 43767 -R /opt/scratch/cwshugg/fall2021/resea
7d4ed97739de846484fd56b1cee05/sub_1638926482/src/fuzz_root  < ../fuzz/fuzz2/cr
2,ss_chunk_havoc
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
Using port 43767
[New Thread 0x7ffff6624700 (LWP 131620)]
[New Thread 0x7ffff5e23700 (LWP 131621)]
Accepted connection from ::1:43240
[New Thread 0x7ffff5622700 (LWP 131622)]

Thread 4 "server" received signal SIGSEGV, Segmentation fault.
[Switching to Thread 0x7ffff5622700 (LWP 131622)]
0x00007ffff6b6cf58 in realloc () from /lib64/libc.so.6
Missing separate debuginfos, use: yum debuginfo-install openssl-libs-1.1.1k-6.
(gdb) backtrace
#0  0x00007ffff6b6cf58 in realloc () from /lib64/libc.so.6
#1  0x00000000004022db in buffer_ensure_capacity (buf=0x7ffff5620d00, len=12)
#2  0x0000000000402348 in buffer_append (buf=0x7ffff5620d00, mem=0x4046c7, len
#3  0x00000000004023bb in buffer_appends (buf=0x7ffff5620d00, str=0x4046c7 "Co
#4  0x0000000000402895 in http_add_header (resp=0x7ffff5620d00, key=0x4046c7 "
.c:168
#5  0x0000000000402db0 in send_error (ta=0x7ffff5620ce0, status=HTTP_BAD_REQUE
(gdb) frame 1
#1  0x00000000004022db in buffer_ensure_capacity (buf=0x7ffff5620d00, len=12)
76              buf->buf = realloc(buf->buf, cap);
(gdb) print buf->buf
$1 = 0x54534f50 <error: Cannot access memory at address 0x54534f50>
```

Figure 4.19: Example Bug 2 - The web server's postmortem stacktrace (GDB).

The participant may next place an appropriate breakpoint and spend time debugging further. This reveals that the server failed to sanitize the `Content-Length` header and stored the negative value (-47) as-is. This is shown in Figure 4.20.

```
131             if (!strcasecmp(field_name, "Content-Length")) {
(gdb) next
132                 ta->req_content_len = atoi(field_value);
(gdb) next
(gdb) print ta->req_content_len
$2 = -47
```

Figure 4.20: Example Bug 2 - Viewing the parsed `Content-Length` header as -47 (GDB).

Through further debugging, the participant discovers the negative `Content-Length` (-47) is used to initialize a stack-local buffer. In a subsequent call to `snprintf()`, the pointer later passed into `realloc()` is overwritten due to the buffer underwrite that occurs. This is shown in Figure 4.21. The root cause of this bug was the participant's failure to sanitize the client's negative `Content-Length` header.

```
Thread 4 "server" hit Breakpoint 2, handle_api (ta=0x7ffff5621e60) at http.c:455
455                 snprintf(
(gdb) list
450             }
451             //client is sending username and password
452             else if (ta->req_method == HTTP_POST) {
453                 //converts the request body into a json object and reads the info
454                 char req_body[ta->req_content_len + 1];
455                 snprintf(
456                     req_body, ta->req_content_len + 1, "%s",
457                     bufio_offset2ptr(ta->client->bufio, ta->req_body)
458                 );
459                 json_error_t error;
(gdb) print ta->resp_headers
$3 = {buf = 0x7fffec002ba0 "Server: CS3214-Personal-Server\r\n", len = 32, cap = 4096}
(gdb) next
460                 json_t *auth_info = json_loads(req_body, 0, &error);
(gdb) print ta->resp_headers
$4 = {buf = 0x0, len = -134225424, cap = 32767}
```

Figure 4.21: Example Bug 2 - Witnessing a stack-local variable be overwritten in an out-of-bounds write (GDB).

Bug 3 - Out-of-Bounds Read - Infinite Loop

The final bug forces the target web server to infinitely read memory until encountering a forbidden address, causing a segmentation fault. As such, it falls under the third major bug category: **out-of-bounds reads**. The participant may first examine the contents of the crash-inducing `comux` file with the provided `comux` toolkit. This `comux` represents a *single* connection across a *single* chunk (shown in Figure 4.22). The chunk contains a HTTP `GET` request for `/private/secure.html`. `Gurthang` mutated the second field within the `Cookie` header to replace the equals sign (`"="`) with the letter `"d"`.

```
* COMUX [version: 0] [num_connections: 1] [num_chunks: 1]
* CHUNK 0: conn_id=0, data_length=247, scheduling=0, flags=0x1
GET /private/secure.html HTTP/1.1
Host: linden.rlogin:27363
User-Agent: python-requests/2.20.0
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive
Cookie: a_bad_cookie1=chocolate_chip; a_bad_cookie2doatmeal_raisin
```

Figure 4.22: Example Bug 3 - The `comux` input file generated by `Gurthang`.

The participant utilizes the provided scripts to re-run the server with the `comux` file to confirm the segmentation fault. This is shown in Figure 4.23.

```
$ ../fuzz/fuzz-rerun.sh ../fuzz/fuzz0/crashes/id:000000*
Using port 13650
Waiting for client... 0
Accepted connection from ::1:50000
Waiting for client... 1
Segmentation fault (core dumped)
```

Figure 4.23: Example Bug 3 - The target web server crashing when given the input file.

Next, the participant launches the GNU Debugger to investigate further [10]. The stacktrace at the time of the crash shows the server failed whilst processing the HTTP request headers. This is shown in Figure 4.24.

```
$ ../fuzz/fuzz-rerun-gdb.sh ../fuzz/fuzz0/crashes/id:000000*
Reading symbols from /opt/scratch/cwshugg/fall2021/research_data/data/116b13dc2
936dd9e/sub_1638749020/src/server...done.
fuzz-rerun-gdb.sh: Arguments are set. Type 'run' to reproduce the crash/hang.
(gdb) run
Starting program: /opt/scratch/cwshugg/fall2021/research_data/data/116b13dc238e
dd9e/sub_1638749020/src/server -p 49281 -R /opt/scratch/cwshugg/fall2021/resear
217b4724f7035e8f89e5a1936dd9e/sub_1638749020/src/fuzz_root  < ../fuzz/fuzz0/cra
3,ss_chunk_havoc
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
Using port 49281
Waiting for client... 0
[New Thread 0x7ffff6624700 (LWP 132794)]
[New Thread 0x7ffff5e23700 (LWP 132795)]
Accepted connection from ::1:48686
[New Thread 0x7ffff5622700 (LWP 132796)]
Waiting for client... 1

Thread 4 "server" received signal SIGSEGV, Segmentation fault.
[Switching to Thread 0x7ffff5622700 (LWP 132796)]
http_process_headers (ta=<optimized out>) at http.c:137
137                     while(*cur != '=') {
Missing separate debuginfos, use: yum debuginfo-install openssl-libs-1.1.1k-6.e
(gdb) backtrace
#0  http_process_headers (ta=<optimized out>) at http.c:137
#1  http_handle_transaction (self=self@entry=0x7ffff5621ee8) at http.c:595
#2  0x00000000004019aa in worker_thread (cs=<optimized out>) at main.c:57
#3  0x00007ffff6e9e1ca in start_thread () from /lib64/libpthread.so.0
#4  0x00007ffff6b09d83 in clone () from /lib64/libc.so.6
```

Figure 4.24: Example Bug 3 - The web server's postmortem stacktrace (GDB).

The participant may then place an appropriate breakpoint and debug further. This reveals that the `Cookie` header parsing code is the point of failure. The participant's code uses an infinite `while` loop to search for an equals sign (`"="`) within the `Cookie` header's contents. The root cause of the bug is the participant's failure to consider the case where *no* equals

sign is present in a malformed `Cookie` header. This is shown in Figure 4.25.

```
(gdb) list
132                         }
133                         while(*cur == ' ') {
134                             cur++;
135                         }
136                         ta->cookie_name[count] = cur;
137                         while(*cur != '=') {
138                             cur++;
139                         }
140                         cur++;
141                         *(cur-1) = '\0';
(gdb) print cur
$3 = 0x7fffec000c63 "a_bad_cookie2doatmeal_raisin\r"
(gdb) c
Continuing.

Thread 4 "server" received signal SIGSEGV, Segmentation fault.
http_process_headers (ta=<optimized out>) at http.c:137
137                         while(*cur != '=') {
(gdb) print (size_t) (cur - field_value)
$4 = 131968
```

Figure 4.25: Example Bug 3 - The faulty source code causes an invalid read far from the original location.

At the time of the crash, this infinite loop walked 132,000 bytes away from its original location. This triggers a segmentation fault.

4.2.7 Participant Bug Fixes over Multiple Fuzzing Campaigns

Many of the study participants chose to submit multiple versions of their web servers across multiple fuzzing attempts. With these multiple submissions, we observed how some students

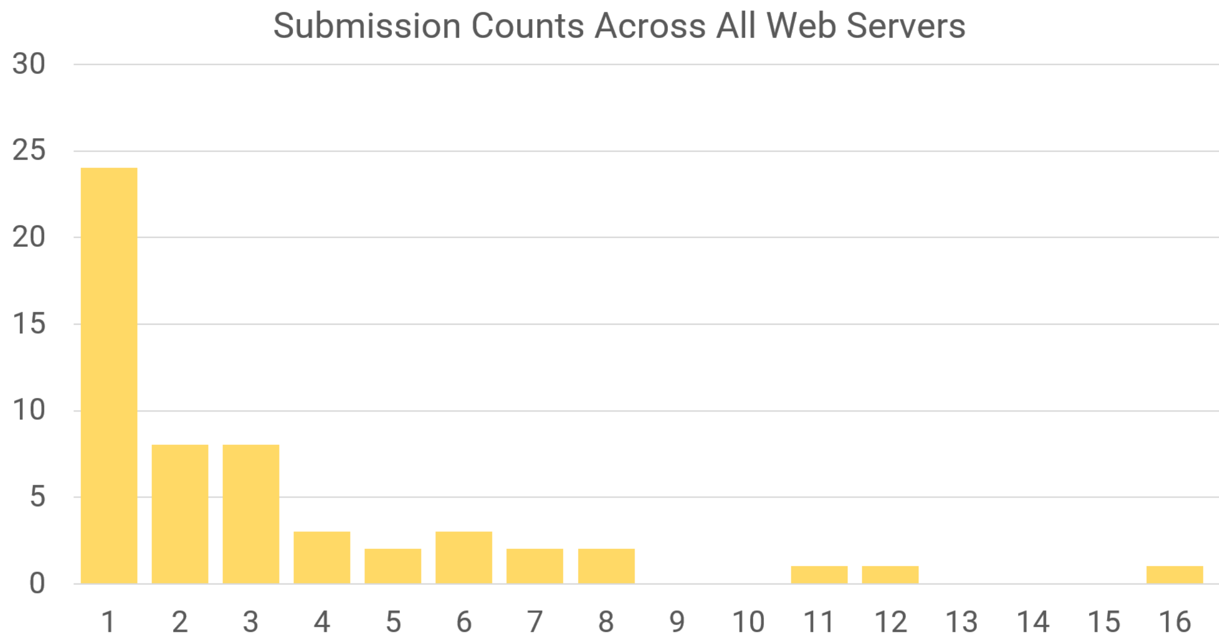

Figure 4.26: A histogram of the number of submissions made across the 55 web servers.

improved their web servers between their fuzzing campaigns.

As shown in Figure 4.26, we observed that the majority of participants only made one or two submissions (corresponding to one or two fuzzing campaigns). Across the 55 distinct web servers, 31 of them were submitted multiple times. 23 of these 31 multi-submitted servers contained bugs found by the fuzzer, and of these, we found that some showcased *different* or *no* bugs in subsequent fuzzing runs. We analyzed these servers to understand how each participant used `Gurthang`'s output to improve their server over time.

Among these 23 buggy & multi-submitted servers, we observed 12 that had *all* previous bugs fixed at the time of their final submission. By examining the source code of these 12 servers across the multiple submissions, we found changes each participant made to fix bugs found in previous fuzzing runs. These observations are proof that some participants that chose to perform multiple fuzzing campaigns spent time debugging and fixing each bug presented to them by the fuzzer, thus making their server more robust in the process. This shows

Gurthang can be easily integrated into the development workflow of developers with little or no experience with fuzzing. We discuss an example from one of the submitted servers below.

Example - Fixing Bugs Over Time

In one particular case, one of these 55 servers exhibited a **failure to check return value** bug. This bug originated from the program attempting to parse a Cookie HTTP request header that Gurthang mutated to be syntactically incorrect. Figure 4.27 shows the HTTP request message enclosed in the crash-inducing comux file.

```
* COMUX [version: 0] [num_connections: 1] [num_chunks: 1]
* CHUNK 0: conn_id=0, data_length=247, scheduling=0, flags=0x1
GET /private/secure.html HTTP/1.1
Host: linden.rlogin:27363
User-Agent: python-requests/2.20.0
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive
Cookie: a_bad_cookie1=chocolate_chip; a_bad_co
```

Figure 4.27: Bug Fix Example - The crash-inducing comux file contains a malformed Cookie HTTP header.

GDB confirmed a crash on line 250 in http.c when given the comux file shown in Figure 4.28 as input.

```
$ ../fuzz/fuzz-rerun-gdb.sh ../fuzz/fuzz0/crashes/id:000000*
Reading symbols from /opt/scratch/cwshugg/fall2021/research_data/data/a27b22936a7c
54488fc/sub_1639083007/src/server...done.
fuzz-rerun-gdb.sh: Arguments are set. Type 'run' to reproduce the crash/hang.
(gdb) run
Starting program: /opt/scratch/cwshugg/fall2021/research_data/data/a27b22936a7ce4f
88fc/sub_1639083007/src/server -p 51833 -R /opt/scratch/cwshugg/fall2021/research_
8534638e1e461a48c3b38454488fc/sub_1639083007/src/fuzz_root  < ../fuzz/fuzz0/crashe
4,ss_chunk_havoc
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
Using port 51833
[New Thread 0x7ffff6624700 (LWP 134361)]
[New Thread 0x7ffff5e23700 (LWP 134362)]
Accepted connection from ::1:41440
[New Thread 0x7ffff5622700 (LWP 134363)]

Thread 4 "server" received signal SIGSEGV, Segmentation fault.
[Switching to Thread 0x7ffff5622700 (LWP 134363)]
0x00007ffff6b9cc95 in __strlen_avx2 () from /lib64/libc.so.6
Missing separate debuginfos, use: yum debuginfo-install openssl-libs-1.1.1k-6.el8.
(gdb) backtrace
#0  0x00007ffff6b9cc95 in __strlen_avx2 () from /lib64/libc.so.6
#1  0x00007ffff6b704e2 in strdup () from /lib64/libc.so.6
#2  0x0000000000403cfb in http_process_headers (ta=<optimized out>) at http.c:250
#3  http_handle_transaction (self=self@entry=0x7ffff5621ee8) at http.c:978
#4  0x0000000000401e8f in helper (client_pm=0x608cc0) at main.c:83
#5  0x00007ffff6e9e1ca in start_thread () from /lib64/libpthread.so.0
#6  0x00007ffff6b09d83 in clone () from /lib64/libc.so.6
```

Figure 4.28: Bug Fix Example - Running the web server through GDB shows the location of the crash.

Examining the source code (shown in Figure 4.29) shows the root cause: the participant failed to check the return value from `strtok_r()`.

```
243             if (strstr(field_value, ";"))
244             {
245                 char *saveptr;
246                 strtok_r(field_value, ";", &saveptr);
247                 char *p = strtok_r(NULL, "; ", &saveptr);
248                 strtok_r(p, "=", &saveptr);
249                 char *q = strtok_r(NULL, "=", &saveptr);
250                 ta->auth_cookie = strdup(q);
251             }
252             else
253             {
254                 ta->auth_cookie = strdup(field_value + strlen(AUTH_TOKEN));
255             }
```

Figure 4.29: Bug Fix Example - The source code reveals the bug: the failure to check for a `NULL` return value from `strtok_r()`.

This server was fuzzed 8 times. Newer versions of the server *did not* exhibit the same crash when provided with the original `comux` file. (Shown in Figure 4.30.)

```
$ ../fuzz/fuzz-rerun-gdb.sh ../../sub_1639083007/fuzz/fuzz0/crashes/id:000000*
Reading symbols from /opt/scratch/cwshugg/fall2021/research_data/data/a27b2293
54488fc/sub_1639087483/src/server...done.
fuzz-rerun-gdb.sh: Arguments are set. Type 'run' to reproduce the crash/hang.
(gdb) run
Starting program: /opt/scratch/cwshugg/fall2021/research_data/data/a27b22936a7
88fc/sub_1639087483/src/server -p 49641 -R /opt/scratch/cwshugg/fall2021/resea
8534638e1e461a48c3b38454488fc/sub_1639087483/src/fuzz_root  < ../../sub_163908
94+000063,time:6044,ss_chunk_havoc
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
Using port 49641
[New Thread 0x7ffff6624700 (LWP 134788)]
[New Thread 0x7ffff5e23700 (LWP 134789)]
Accepted connection from ::1:47164
[New Thread 0x7ffff5622700 (LWP 134790)]
HTTP/1.1 403 Permission Denied
Server: CS3214-Personal-Server
Content-Type: application/json
Content-Length: 2

{}
[Thread 0x7ffff5622700 (LWP 134790) exited]
[Thread 0x7ffff5e23700 (LWP 134789) exited]
[Thread 0x7ffff6624700 (LWP 134788) exited]
[Inferior 1 (process 134784) exited normally]
Missing separate debuginfos, use: yum debuginfo-install openssl-libs-1.1.1k-6.
```

Figure 4.30: Bug Fix Example - Running the same input file with a *newer* version of the same web server shows it no longer crashes.

Examining the same area of source code in the newer submission (`http.c` line 250), shows the participant added an if-statement to check against a `NULL` return value, thus fixing the bug. (Shown in figure 4.31.)

```
243             if (strstr(field_value, ";"))
244             {
245                 char *saveptr;
246                 strtok_r(field_value, ";", &saveptr);
247                 char *p = strtok_r(NULL, "; ", &saveptr);
248                 strtok_r(p, "=", &saveptr);
249                 char *q = strtok_r(NULL, "=", &saveptr);
250                 if (q)
251                     ta->auth_cookie = strdup(q);
252             }
253             else
254             {
255                 ta->auth_cookie = strdup(field_value + AUTH_LEN);
256             }
```

Figure 4.31: Bug Fix Example - The newer submission's source code shows the participant added a return value check to fix the bug.

This example (and others we observed) showcases `Gurthang`'s ease-of-use, considering these students had little (or *no*) experience in writing HTTP web servers and fuzzing programs. Despite this inexperience, students were easily able to integrate fuzzing into their software development process to make their code more robust.

4.3 Survey Results

In addition to collecting and analyzing participants' source code and fuzzer output, we invited students to fill out an anonymous Qualtrics survey to provide feedback on the fuzzing tools' effectiveness. The survey was opened after IRB approval was granted, and the research study was made public to the students in CS 3214. 16 students responded to the survey. We list the questions that made up the survey below, with a count listed next to each possible response, indicating the responses that were collected.

1. **On a scale of 1-5, how much would you say you knew about fuzzing before using this tool?**

   - One..[12|############]
   - Two..[3|###]
   - Three......................................[0|]
   - Four.......................................[1|#]
   - Five.......................................[0|]

2. **On a scale of 1-5, how much would you say you know about fuzzing NOW, after using this tool?**

   - One..[0|]
   - Two..[3|###]
   - Three......................................[10|##########]
   - Four.......................................[3|###]
   - Five.......................................[0|]

3. **Do you think this tool helped you achieve better results on project 4?**

   - The tool helped significantly................[3|###]
   - The tool helped slightly.....................[9|#########]
   - The tool didn't help at all..................[4|####]
   - The tool was counterproductive...............[0|]

4. **Overall, how useful did you find the tool to be?**

   - Very useful................................[10|##########]

- `Slightly useful`.............................[4|####]
- `Not useful at all`............................[2|##]
- `Counterproductive`............................[0|]

5. **Do you believe this tool reported any inaccuracies?**
   - `Yes`...[1|#]
   - `No`..[15|###############]

6. **Is there anything else you would like to share about the tool?**
   - `Yes`...[8|########]
   - `No`..[8|########]

Our final question prompted participants to provide open-ended feedback about anything regarding `Gurthang` and its usage in CS 3214. 8 of the 16 respondents left comments. They are listed below.

- *Good work!*
- *Great UI, and output to showcase how we can recreate the bug in GDB.*
- *I think it should be integrated into the grading tool so it's easier to use.*
- *It has a great UI, and it was easy to use without having to specify or configure a bunch of other things.*
- *It was super nice to get extra credit and seeing no bugs was a nice way of confirming that our server was operating at a strong level.*
- *Loved the documentation and how easy it was to use. Great UX.*

- *Really cool project Connor!*
- *The only reason I said the fuzzer was not helpful was that we only ran it after we passed all of the other tests, so it ran successfully the first time and reported zero crashes. I'm sure that if we had run it earlier in development it would have been helpful in finding bugs.*

We made an extra credit opportunity available to every student in CS 3214, regardless of participation in the research study. To avoid bias, we offered this extra credit to participating *and* non-participating students. We were advised by the IRB to leave the extra credit opportunity out of the official research protocol.

Despite the relatively small number of responses, the survey shows largely positive feedback on `Gurthang` and its effectiveness in finding bugs in the students' code. The first two questions show that participants believed their knowledge of fuzzing was increased after using `Gurthang` to fuzz their web server. The majority of respondents also claimed the tool helped them achieve better results, if only slightly, on their projects. Finally, the majority also claimed the tool to be "very useful," and did not report any inaccuracies.

It is through these survey responses that we can conclude an overall positive effect and an achievement of one of our goals in conducting this study in CS 3214: providing students with a hands-on example of fuzzing and introducing them to a new field in Computer Security.

4.4 Fuzzing Real-World Web Servers

While we largely focused our evaluation on Fall 2021 CS 3214 research study, we also spent time fuzzing two industry standard open-source web servers to evaluate `Gurthang` further: **Apache** and **Nginx**.

We performed this evaluation on a standalone Linux machine running Ubuntu version 18 with the following hardware specifications:

- **Machine:** Dell Optiplex 7010
- **Processors:** 4x Intel Core i5-3475S 2.90GHz 4-core processors
- **Memory:** 8 GB RAM
- **Disk:** 1 TB hard drive
- **Ethernet:** 1 Gigabit interconnect

4.4.1 Fuzzing Apache

The Apache HTTP server project is the most widely-used HTTP server in the world [14]. As such, we selected Apache as our first real-world target.

Configuring Apache

To evaluate `Gurthang`'s ability to fuzz Apache, we downloaded version 2.4.51 of its source code to our system and configured it with the following settings:

- We pointed the `CC` and `CXX` environment variables at `AFL++`'s `afl-clang-fast` compiler.
- We force-added debugging symbols to the compilation stage by adding `-g` to the `CFLAGS`, `EXTRA_CFLAGS`, and `CPPFLAGS` build variables. We added this to enable easier debugging in the event `Gurthang` discovered a crashing test case.

- We installed Apache to a local directory using the `--prefix` configure setting. This was done to avoid the need for root permission on our Linux system.
- We enabled the `--with-included-apr`, `--enable-static-support`, `--enable-mods-static=few`, `--disable-pie`, `--enable-debugger-mode`, and `--with-mpm=worker` settings to reduce number of modules enabled in Apache, enable debugging mode for easier fuzzing, and specify our desired threading model.

During the invocation to `make` to build Apache, we set the `AFL_LLVM_LAF_ALL` environment variable to enable the `AFL++` compiler to split up comparison operations into multiple smaller ones, making path discovery easier for the fuzzer [9, 22].

We selected a threading model that spawned four concurrent worker threads under a single process. This choice enables `Gurthang` to exercise multiple connections, handled across Apache's multiple worker threads. Apache has a number of static modules (shared libraries) that add various features (such as authentication or caching) to the web server. These are linked with Apache at compile-time. We chose to use `few` static modules in order to focus on fuzzing Apache's core static-file-serving functionality. While this is a heavily tried-and-tested portion of Apache's features, we believed it served as a reliable starting point.

We configured Apache to bind to an unprivileged port number and listen on a local IP address. We also configured the server to run on a single process that spawned four separate worker threads.

Fuzzing Setup

We used the same set of input files for the CS 3214 study to fuzz Apache. While these files originally contained raw HTTP request messages, we converted them into `comux` files to be compatible with `Gurthang`. In addition to this, we created a root directory containing

several sub-directories and files of varying types to make the discovery of different static file types easier for `Gurthang`. The varying file types consisted of plain text documents (`.txt`), HTML files (`.html`), PDFs (`.pdf`), images (`.png`), icon files (`.ico`), MP4 videos (`.mp4`), comma-separated files (`.csv`), Cascading Style Sheets (`.css`), and JSON files (`.json`). We gave each sub-directory and file a very short file name (or a common name) to make it easy for the fuzzer to request new files in HTTP `GET` requests, potentially invoking new execution behavior in Apache. Examples of these file names include: `a`, `b`, `0`, `1`, `_`, `a.html`, `public/`, `private/`, and `index.html`.

We also created three dictionaries containing HTTP header names, header values, and request methods, to enable `Gurthang`'s dictionary-swap mutation using HTTP keywords.

Fuzzing Results

When running the fuzzer in a few short trial runs, we observed that `Gurthang` was capable of fuzzing Apache without modifying a single line of its source code. Initially, the fuzzer achieved around 65 executions per second. We suspected `AFL++` could be sped up by using its deferred initialization to delay the creation of `AFL++`'s internal fork server until *after* Apache's initialization was complete [13]. Deploying this increased `AFL++`'s execution speed eightfold, up to 500-600 executions per second.

In this improved state, we ran `AFL++` for 21 CPU-days. No bugs were discovered. However, `Gurthang` successfully enabled `AFL++` to discover many new execution paths in Apache. The initial set of input files we provided to `AFL++` yielded 516 initial unique execution paths within Apache. Throughout the 21-day fuzzing campaign, the fuzzer discovered 2491 additional unique paths.

4.4.2 Fuzzing Nginx

Like Apache, Nginx is an extensive, open-source HTTP server that supports similar features and multi-threading through the use of thread pools. [48]. We chose Nginx as the second real-world target.

Configuring Nginx

We followed a similar process to configure and build Nginx as we did with Apache. We downloaded Nginx's source code (version 1.21.6) to our system, then configured with the following settings:

- We installed Nginx to a non-privileged directory using the `--prefix` and `--sbin-path` options.
- We downloaded the source code for PCRE (one of Nginx's dependencies) and pointed its location to the `--with-pcre` setting.
- We downloaded the Zlib source code (another dependency) and pointed its location to the `--with-zlib` setting. (We later discovered this to be unnecessary, as PCRE and Zlib could instead be disabled.)
- We enabled the Nginx debug log via the `--with-debug` setting.
- We ensured the inclusion of debug symbols during compilation by adding the `-g` flag using the `--with-cc-opt` setting.
- We enabled Nginx's threadpool by enabling the `--with-threads` setting.

We chose the above settings from the same motivations that drove our decisions while fuzzing Apache: to make debugging any crashing test cases easier and enable the fuzzing of multiple connections.

We enabled only a small number of modules, just as we did with Apache, to focus on fuzzing Nginx's core static-file-serving functionality. The following module settings were used:

- `--with-http_v2_module` enabled support for HTTP/2.
- `--with-http_realip_module` enabled the changing of the client address to be the address sent in an HTTP request's header field.
- `--with-http_mp4_module` enabled streaming support for MP4 files.
- `--with-http_auth_request_module` enabled client authentication requests.
- `--with-http_slice_module` enabled the splitting and caching of HTTP request messages.
- `--with-http_stub_status_module` enabled support for requests asking for server status information.

As done with Apache, we pointed the `CC` environment variable at `AFL++`'s `afl-clang-fast` compiler and enabled the `AFL_LLVM_LAF_ALL` environment variable [9, 22]. Additionally, we configured Nginx run as a standard Linux process (i.e. *not* a daemon) that spawned eight worker threads and listened for connections on an unprivileged port number.

Fuzzing Setup

We used the exact same fuzzing setup Nginx as described in Section 4.4.1 for Apache. We created the same fuzzing-friendly root directory for Nginx to serve files from, and we used

the same input corpus of `comux` files as the initial seed for `AFL++`.

Fuzzing Results

Just as observed with Apache, `Gurthang` allowed `AFL++` to fuzz Nginx without any modification of Nginx's source code. Initially, `AFL++` achieved around 200 executions per second. We increased this speed up to 500 executions per second by deploying `AFL++`'s deferred initialization [13]. We fuzzed Nginx for 21 CPU-days. No bugs were discovered. However, we observed that `Gurthang` enabled `AFL++` to discover several unique execution paths throughout the 21-day campaign. The initial input corpus provided `AFL++` with 516 initial unique paths (just like Apache). Throughout the campaign, `Gurthang` enabled `AFL++` to discover 1489 additional unique execution paths.

4.5 Evaluation Results

In this section we review the evaluation goals established in Section 4.1 to discuss the lengths at which they have been met.

Goal 1 - Easy Integration with State-of-the-Art Fuzzers

We built `Gurthang`'s custom mutator as an `AFL++` custom mutator. Because of this, `Gurthang` can be easily integrated into `AFL++`. We have shown this by harnessing `AFL++` with `Gurthang` to fuzz a large number of distinct web servers.

Additionally, our combination of the `comux` file format with `Gurthang`'s `LD_PRELOAD` library enables web servers to accept input through network connections managed by the input bridge. We were successful in fuzzing *all* of the web servers used in our evaluation with *no*

source code modification required.

Goal 2 - Ability to Fuzz a Variety of Web Servers

We believe `Gurthang` is very effective at fuzzing a variety of web servers in two primary ways:

Zero source code modification is required. We designed and implemented `Gurthang`'s input bridge to forward the contents of the standard input stream to a web server through a network connection without modifying a single line of the server's source code. In our evaluation, we showed this by fuzzing 55 distinct web servers from our research study, as well as Apache and Nginx, without modifying *any* source code. Given the varying implementation of the CS 3214 web servers and the complexity of Apache and Nginx, we conclude `Gurthang` to be a success on this front.

`Gurthang` can fuzz a variety of web server implementations. During our Fall 2021 CS 3214 study, students implementing their web servers chose a variety of multi-threading models. Different multi-threading models required the use of different system calls (such as `accept()`, `select()`, or `epoll_wait()`) to await new client connections. We adapted our implementation of `Gurthang` to handle all unique approaches, allowing us to fuzz all 55 web servers. Apache and Nginx took some initial configuration to use the desired threading models, but `Gurthang` fuzzed and exercised multiple connections for both of them without any additional change.

Students in CS 3214 implemented a wide variety of features into their web servers, as discussed in section 4.2.1. Each distinct server took different approaches to all features. Despite these differences, `Gurthang` was capable of fuzzing every implementation.

Goal 3 - Discovering Bugs

As we discussed in Section 4.2.5, We fuzzed 55 distinct web servers and discovered 48 total bugs of varying cause. During fuzzing, we enabled `AFL++`'s `AFL_CUSTOM_MUTATOR_ONLY` environment variable, which disables *all* `AFL++`-built-in fuzzing mutations and relies solely on the mutations built into `Gurthang`'s [9]. This means `AFL++` was not responsible for mutating the `comux` files; `Gurthang` performed all mutations and thus created all test cases during our fuzzing campaigns. These test cases eventually enabled `AFL++` to discover crashes in the web servers.

Goal 4 - Going Beyond Existing Tools

Our results from the CS 3214 research study show that the bugs we discovered evaded the detection of the existing unit test suite. Our coverage measurements described in Section 4.2.4 show that the unit tests exercise a sufficient amount of each web server's functionality. Despite many servers passing these unit tests prior to fuzzing, `Gurthang` still discovered bugs in them. With this, we can conclude that fuzzing was necessary for the discovery of these bugs, and `Gurthang` enabled their discovery.

It is possible other state-of-the-art testing tools, such as Valgrind's `memcheck` utility and LLVM's Undefined Behavior Sanitizer would have assisted the participants in uncovering the root causes of these bugs prior to discovering them with the fuzzer [23, 34, 50]. When analyzing the bugs discovered by `Gurthang`, we confirmed these tools do indeed detect them once the web server has been fed the fuzzer's input file. However, without the input file crafted by the fuzzer, the web server likely would not have exhibited any faulty behavior during normal testing. This further emphasizes our conclusion: the bug would not have surfaced under normal testing, and thus fuzzing was needed to discover these bugs.

4.6 Evaluation Limitations

We made certain choices during evaluation that introduced some limitations. This section discusses those limitations.

A separate assignment within CS 3214 in the Fall 2021 semester required students to implement MP4 video streaming onto their servers. At the time of the study, the unit tests provided to students did *not* exercise the video streaming functionality. However, a few test cases we seeded `AFL++` with attempted to invoke this relatively untested functionality. Three bugs discovered during the study exploited the video streaming functionality in some servers; these bugs may have been fixed prior to fuzzing if appropriate unit tests existed. However, the majority of bugs we discovered (the remaining 45) did not involve this shortcoming. (As of April 2022, we have implemented unit tests to exercise this functionality.)

Additionally, at the time of the study, we had not yet implemented the `CHUNK_DICT_SWAP` mutation in `Gurthang`'s mutator, and thus the use of HTTP dictionaries was not a part of this portion of our evaluation. However, the web servers implemented in CS 3214 use a very small subset of HTTP request methods and headers, meaning the use of HTTP dictionaries may not have had much additional effect.

We fuzzed Apache and Nginx's basic static file serving functionality. A small number of additional modules were enabled in both servers during fuzzing, but we provided no specific test cases to `AFL++` that invoked them. In doing this we limited `Gurthang` to fuzzing a robust and relatively simple portion of the two servers' overall behavior. It is possible `Gurthang` could have made more discoveries if we spent time fuzzing other parts of Apache and Nginx, such as configuration files, authentication functionality, cache behavior, and other features built into both servers.

We sought to address the challenge of fuzzing a web server's ability to properly identify

message boundaries when reading a HTTP messages through a network socket, as described in Section 1.1. All 55 web servers from the CS 3214 study were built atop the same set of starter code that implemented this functionality without flaw. As such, our evaluations did not thoroughly test a *diverse* set of message boundary functionality.

Finally, `AFL++` has many configurable features that can assist with discovering bugs in certain contexts [9]. One example is the integration of power schedules originating from AFLFast that steer test case creation towards new execution behavior faster than the original `AFL` [4]. We largely left `AFL++` with its default settings and did not take advantage of the full feature set. By doing this, we may have limited `Gurthang`'s ability to discover bugs.

Chapter 5

Related Work

In this chapter we discuss previous research and `Gurthang`'s relationship to it.

5.1 Related Work in Fuzzing

Researchers have explored various aspects of fuzzing in order to better test certain target programs. We discuss some of these pursuits in this section.

Grammar Fuzzing

Grammar-based fuzzers are designed to be fully aware of the target program's input format [2, 46]. This enables them to generate diverse test cases that pass the target's syntactic and semantic checks to exercise deeper paths of execution. Grammar fuzzers typically use Context-Free Grammar specifications and parse trees to perform grammar-friendly test case mutation and generation. Two such grammar-based fuzzers, Nautilus and Gramatron, have been successful at finding bugs deep within certain language interpreters [2, 46]. In comparison, `Gurthang` is not aware of the grammar of its target programs (HTTP, FTP, and other networking protocols) and thus cannot reap the benefits of grammar fuzzing. However, `Gurthang` is more general-purpose, allowing it to more easily test a diverse set of targets.

Hybrid Fuzzing

Hybrid fuzzing utilizes both traditional fuzzing techniques and **concolic execution** (a combination of **symbolic execution** and **concrete testing**) to combine their strengths and mitigate their individual weaknesses. Two of these hybrid fuzzers are Driller [47] and QSYM [56]. They use concolic execution to steer fuzzing into new compartments of the target's behavior guarded by specific conditions (such as an if-statement checking for `0x00abcdef`). Once new behavior is found through concolic execution, general-purpose fuzzing takes over until the search space is relatively exhausted. This strategy yields a variety of areas within the target program to explore, and it has been shown to discover previously-unknown bugs in real-world programs [56]. `Gurthang` harnesses `AFL++`, which does not use concolic execution in any way. However, improving `Gurthang` to use a hybrid fuzzer may be beneficial as future work.

Binary-Only Fuzzing

Despite the heavy usage and high success rate of source-code-guided fuzzing, situations still exist where source code is not available for a target program. Binary-only fuzzing seeks to transform an executable to create a more effective feedback loop while maintaining the performance of the original binary. ZAFL, one such binary-only fuzzer, re-writes the target program's binary to inject inlined instrumentation that achieves similar feedback and performance to traditional compile-time instrumentation seen from grey-box fuzzers like `AFL` and `AFL++` [33]. `Gurthang` was implemented to harness `AFL++`, which requires open source code. However, no part of `Gurthang` *depends* on this, meaning it could be modified to harness a binary-only fuzzer to enable the fuzzing of closed-source web servers. We leave this as future work.

Fuzzing Harnesses

Fuzzing harnesses bridge the gap between a specific target program and a fuzzer. (`Gurthang`'s `LD_PRELOAD` library is a fuzzing harness for web servers.) Harnesses strive for compatibility to make more target programs available for fuzzing. For example: kAFL, an `AFL`-like fuzzer designed to find bugs in kernel execution, implements a harness that wraps the target kernel in a Virtual Machine and executes a user-space program that invokes the kernel with the fuzzer's test case [43].

`Gurthang` is predated by Preeny, an open-source project containing a variety of `LD_PRELOAD` libraries to modify target program behavior [45]. One of Preeny's libraries acts as a fuzzing harness for network applications by "de-socketing" a web server (i.e. replacing the use of socket file descriptors with standard file streams). Similar to `Gurthang`'s `LD_PRELOAD` library, Preeny's "de-socketer" uses internal threads to intercept network programming system calls. However, rather than establish local connections to the target server (like `Gurthang`), Preeny instead mimics a subset of the socket programming API by interacting with `stdin` and `stdout`. Preeny's approach is functional, but it has some drawbacks we address with `Gurthang`:

- Preeny's "de-socketing" approach only works when the target program makes the `read()` and `write()` system calls. Typically, web servers use the `recv()` and `send()` calls to interact with socket file descriptors. Because of this, source code modification may be necessary when fuzzing with Preeny.
- Preeny is not thread-safe, and thus can only safely fuzz single-threaded web servers.
- Preeny can only manage a single client connection when harnessing web servers.

After the conclusion of `Gurthang`'s development and at the time of writing of this thesis

(the winter and spring of 2022), Libdesock, a new network application fuzzing harness, was created [17]. Libdesock emulates the entire network stack of the Linux kernel entirely within user-space, supports multi-threaded applications, and claims to have a 5x speedup when compared to Preeny [8]. It has been tested against Nginx, Apache, OpenSSH, OpenVPN, and a host of other networking applications.

In terms of efficiently fuzzing networking applications, Libdesock appears to re-define the state-of-the-art. `Gurthang` does *not* emulate the entire networking stack and as such is still subject to the same performance drawbacks as Preeny. However, `Gurthang` still introduces novelty in its connection multiplexing design. Both `Gurthang` and Libdesock support multi-threaded web servers, but `Gurthang` explicitly exercises multiple threads at once through the `comux` file format and `LD_PRELOAD` library's interaction with the server.

5.2 Network Application Fuzzers

A number of networking fuzzers exist. Each take different approaches to fuzzing network applications. We discuss some of them in this section, and how our research relates.

AFLNet

AFLNet is a fork of the original `AFL` designed to fuzz applications that use well-known networking protocols to communicate [39, 40]. It internally connects to the target web server and sends test cases via a socket. AFLNet uses a state machine that parses protocol response codes (such as HTTP response status codes as discussed in Section 2.2.2) to influence test case decisions during a fuzzing campaign. While this state machine is novel, there are drawbacks to AFLNet's implementation:

- When compared to AFL++ (harnessed with Gurthang), AFLNet is much slower. Across several 10-hour trial runs, we observed AFL++ to achieve an average of 379.5% more executions in the same amount of time.

- AFLNet does not support multiple connections to the target web server [40]. Gurthang does, through its LD_PRELOAD library and the comux file format.

While AFLNet contributes a functional and novel approach to fuzzing certain network applications, it was developed prior to AFL++'s creation (which exceeds it in speed and path discovery) and has some design limitations that have caused it to fall behind the state-of-the-art.

AFL++'s Network-Fuzzing Branch

The AFL++ GitHub repository contains a deprecated branch designed for fuzzing network applications [9]. However, the developers have noted that the performance is not adequate when compared to harnessing the standard AFL++ with Preeny's "de-socketing" LD_PRELOAD library.

Gurthang is comparable in performance to Preeny. Preeny itself has some drawbacks that make it an imperfect solution for fuzzing network applications (as we discuss in Section 5.1). This, along with other advantages Gurthang introduces, makes the AFL++ networking branch an undesirable choice. We chose to include it in this section to showcase the fuzzing community's awareness of the need for effective grey-box network fuzzing solutions.

Boofuzz

Boofuzz is an extensive network fuzzer that accepts a user-written protocol format and uses instrumentation similar to `AFL`'s to detect failures in the target web server [37, 38]. Users must create a small Python programs that makes calls to Boofuzz's API to facilitate fuzzing. Although initial setup is more involved when compared to `AFL`-like fuzzers, Boofuzz can understand the grammar of networking protocols at a granularity chosen by the user, making it a grey-box fuzzer *and* a grammar fuzzer, unlike `Gurthang`. It is also capable of playing the role of a HTTP server to fuzz *client* networking applications. Boofuzz was written in Python, and as such we suspect it may be less optimized than `AFL++`, which was written in C. Unfortunately, due to time constraints, we did not evaluate `Gurthang` against Boofuzz.

Wfuzz

Wfuzz is a black-box HTTP-specific fuzzer [30, 31]. Because of its focus on HTTP, it is capable of parsing full HTTP messages and mutating specific fields within HTTP messages to create diverse test cases. It also maintains knowledge of previously-requested server URLs and previously-received response codes, which allows it to direct its efforts towards new behavior. Wfuzz is proficient at understanding HTTP and targeting various aspects of the protocol. However, it is designed to target a *remote* web server, and thus is a black-box fuzzer. It relies solely on the remote server's HTTP responses to form a feedback loop and thus may be outpaced in many cases by grey-box fuzzers such as `AFL++`. `Gurthang`, in comparison, has the advantage of using `AFL++`'s feedback loop and speed. Because of this, we suspect `Gurthang` and `AFL++` will achieve more discovery than Wfuzz. Again, due to time constraints, we did not evaluate `Gurthang` against Wfuzz.

Chapter 6

Future Work

We designed `Gurthang` to easily fuzz web servers with state-of-the-art fuzzing tools and a unique connection multiplexing protocol that exercises multiple connections to the server in a single run of the server. While our evaluation shows it has met our goals, we outline many ways it can be improved in this section.

6.1 Increased Performance

It is possible some related works in network application fuzzing achieve better performance (such as more fuzzer executions per second) than `Gurthang`. Generally, we have observed `Gurthang` to have acceptable performance, but optimizations could be made to the `LD_PRELOAD` library's threading model, the `AFL++` mutator's parsing of `comux` files, and other internal procedures, to improve fuzzing throughput.

For example: `Gurthang` presently must pay the overhead of spawning one thread for *each* chunk in a `comux` file. This overhead increases with the `comux` file's chunk count. A possible optimization would be to instead use fewer threads (perhaps a single thread, or one for each connection) to manage these chunks.

6.2 Protocol Awareness

As we discussed in Chapter 5, a fuzzer that understands the grammar of the target program's inputs can be more capable of uncovering new behavior in the target program during fuzzing by generating test cases that pass the target's syntactic and semantic checks [2, 21, 46].

`Gurthang`'s `AFL++` custom mutator module presently does *not* implement such syntax awareness. In one sense, this is beneficial, as by *not* focusing on one particular networking protocol (such as HTTP), we have created `Gurthang` as a fuzzing framework for *any* networking application. In a different sense, though, we may have held `Gurthang` back by neglecting to implement protocol awareness for HTTP. We leave the implementation of HTTP awareness into `Gurthang` as future work, as it may lead to more execution path discovery (and by extension, more bug discovery) during fuzzing. Of all future works to pursue, we believe this one is the most advantageous.

6.3 True Parallel Communication

In Section 3.3.1 we describe the internal threading model of the `Gurthang` `LD_PRELOAD` library. The controller thread oversees the spawning of secondary chunk threads for each chunk present in the `comux` file it parsed from `stdin`. Before spawning the *next* chunk thread, the controller thread waits for the previous to exit. Even though `Gurthang` enables the ability to have multiple simultaneous connections open with the target web server, truly-parallel communication is not achieved due to this threading model.

In one way, this is advantageous in terms of stability. `AFL++`'s code instrumentation is thread-safe, but threads are scheduled by the Linux kernel based on many factors. These factors change over time, meaning multiple threads will not be scheduled in the same order each

time a program is executed. The fact that `Gurthang`'s `LD_PRELOAD` library threads explicitly wait for the previous threads to complete before spawning new ones imposes an ordering of the threads, *regardless* of how they are scheduled. This means subsequent runs of the same `comux` file under `AFL++` will be relatively stable (i.e. the same exact execution paths will be revealed in `AFL++`'s instrumentation most of the time). Some risk of instability still exists, as we discuss in Section 3.5, but we haven't observed these risks to impede fuzzing.

The downside to this approach is that `Gurthang` does not cause the target server to handle two requests across two separate connections in *true* parallel. This means some bugs that involve multi-threading race conditions (such as two threads racing to write to the same unprotected global variable) may not be revealed by `Gurthang`.

6.4 Further Testing of Real-World Web Servers

As we discuss in Section 4.4, we evaluated `Gurthang` by fuzzing the basic static file serving behavior of both Apache and Nginx. We did not spend time fuzzing other, more complex modules within these servers. In addition, other well-known open-source web servers exist that could be used to evaluate `Gurthang`'s capabilities, such as Hiawatha and OpenLite-Speed [44, 54]. We leave this as future work.

Chapter 7

Conclusions

Many state-of-the-art fuzzers lack the ability to effectively perform fuzz-testing on network applications due to the unique constraints they follow. Past research has attempted to address this issue, but has certain drawbacks that make the fuzzing of network applications difficult.

To address the issue and provide a more capable network fuzzing framework, we designed and developed `Gurthang`, a fuzzing framework that enables the state-of-the-art fuzzer `AFL++` to perform fuzz-testing on single-threaded and multi-threaded web servers across multiple simultaneous socket connections, all without the need for any source code modification. We implemented `Gurthang` with two shared libraries and a unique file format (the `comux` file format). Our implementation allows for a web server to accept test cases from a fuzzer and exercise multiple concurrent client connections through the information specified in each `comux` file. Our mutator library mutates both the data within these `comux` files and the instructions that dictate *how* and *when* this data will be sent to the target server. We accomplish all of this without having to modify the source code of the target web server.

We evaluated `Gurthang` using 55 distinct web servers submitted through a study during the Fall 2021 semester in Virginia Tech's CS 3214: Computer Systems course, as well as on two real-world web servers (Apache and Nginx). With `Gurthang`, we discovered 48 bugs of varying origin on the 55 web servers throughout the duration of the Fall 2021 study. We discovered no bugs in Apache and Nginx's static file serving functionality. Every web server

we used to evaluate `Gurthang` was fuzzed easily and without any source code modification.

As we discuss in Section 1.3, this research makes multiple contributions. `Gurthang` provides a simple approach to fuzzing network applications and introduces novelty in its ability to establish multiple simultaneous connections with the target application in a single execution of a fuzzing campaign. We open-sourced `Gurthang` to encourage its use and further evaluation on GitHub (`https://github.com/cwshugg/gurthang`). Lastly, we contribute our evaluation of `Gurthang` that proves its ability to fuzz web servers and discover bugs.

Bibliography

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, &; Tools*. Pearson Addison-Wesley, 2nd edition, 2007.

[2] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. NAUTILUS: Fishing for deep bugs with grammars. In *Network and Distributed Systems Security (NDSS) Symposium*, San Diego, CA, USA, 01 2019. doi: 10.14722/ndss.2019.23412.

[3] A. Barth. RFC 6265 - HTTP state management mechanism, April 2011. URL `https://datatracker.ietf.org/doc/html/rfc6265`.

[4] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1032–1043. Association for Computing Machinery, 2016. doi: 10.1145/2976749.2978428. URL `https://doi.org/10.1145/2976749.2978428`.

[5] D. Clark. The design philosophy of the DARPA internet protocols. In *Symposium Proceedings on Communications Architectures and Protocols*, SIGCOMM '88, page 106–114. Association for Computing Machinery, 1988. doi: 10.1145/52324.52336. URL `https://doi.org/10.1145/52324.52336`.

[6] Ben Collins. Libjwt - github repository, May 2015. URL `https://github.com/benmcollins/libjwt`. Accessed 2022-2-23.

[7] MITRE Corporation. CWE - common weakness enumeration, Accessed 2022-3-8. URL `https://cwe.mitre.org/`.

[8] Patrick Detering. lolcads tech blog - fuzzing network applications with AFL and libdesock, February 2022. URL `https://lolcads.github.io/posts/2022/02/libdesock/`. Accessed 2022-3-10.

[9] AFL++ Developers. Aflpluslpus - the AFL++ fuzzing framework, Accessed 2022-3-8. URL `https://aflplus.plus/`.

[10] GDB Developers. *GDB: The GNU Project Debugger - Documentation*, Accessed 2022-2-23. URL `https://www.sourceware.org/gdb/`.

[11] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++ : Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, August 2020. URL `https://www.usenix.org/conference/woot20/presentation/fioraldi`.

[12] Andrea Fioraldi, Alessandro Mantovani, Dominik Maier, and Davide Balzarotti. Registered report: Dissecting american fuzzy lop - a fuzzbench evaluation. In *FUZZING 2022, 1st International Fuzzing Workshop, 24 April 2022, San Diego, CA, USA / Co-located with NDSS 2022*, San Diego, 2022.

[13] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++ - github repository, Accessed 2022-2-28. URL `https://github.com/AFLplusplus/AFLplusplus`.

[14] The Apache Software Foundation. Apache HTTP server project, Accessed 2022-3-11. URL `https://httpd.apache.org/`.

[15] The Open Web Application Security Project Foundation. The OWASP foundation, Accessed 2022-3-8. URL https://owasp.org/.

[16] The Open Web Application Security Project Foundation. Vulnerabilities, Accessed 2022-5-17. URL https://owasp.org/www-community/vulnerabilities/.

[17] Information Processing Fraunhofer Institute for Communication and Ergonomics FKIE Cyber Analysis & Defense Department. Libdesock - github repository, Accessed 2022-3-10. URL https://github.com/fkie-cad/libdesock.

[18] Owen Garrett. Inside nginx: How we designed for performance & scale, June 2015. URL https://www.nginx.com/blog/inside-nginx-how-we-designed-for-performance-scale/. Accessed 2022-3-11.

[19] Patrice Godefroid. Fuzzing: Hack, art, and science. *Commun. ACM*, 63(2):70–76, jan 2020. ISSN 0001-0782. doi: 10.1145/3363824. URL https://doi.org/10.1145/3363824.

[20] The Open Group. The open group base specification, 2018. URL https://pubs.opengroup.org/onlinepubs/9699919799.2018edition/.

[21] Shengtuo Hu. AFL++ grammar mutator - github repository, Accessed 2022-2-23. URL https://github.com/AFLplusplus/Grammar-Mutator.

[22] LAF-Intel. LAF-LLVM-Pass - gitlab repository, Accessed 2022-4-5. URL https://gitlab.com/laf-intel/laf-llvm-pass.

[23] Chris Lattner and Vikram 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, Mar 2004.

[24] Petri Lehtinen. Jansson - github repository, Accessed 2022-2-23. URL `https://github.com/akheron/jansson`.

[25] E. Blanton M. Allman, V. Paxson. RFC 5681 - tcp congestion control, September 2009. URL `https://datatracker.ietf.org/doc/html/rfc5681`.

[26] Linux man-pages project. *diff(1) - Linux Programmer's Manual*, Accessed 2022-2-24. URL `https://www.man7.org/linux/man-pages/man1/diff.1.html`.

[27] Linux man-pages project. *signal(7) - Linux Programmer's Manual*, Accessed 2022-3-8. URL `https://www.man7.org/linux/man-pages/man7/signal.7.html`.

[28] Linux man-pages project. *dlsym(3) - Linux Programmer's Manual*, Accessed 2022-3-9. URL `https://man7.org/linux/man-pages/man3/dlsym.3.html`.

[29] Valentin J.M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, 2021. doi: 10.1109/TSE.2019.2946563.

[30] Xavier Mendez. *Wfuzz: The Web fuzzer*, March 2017. URL `https://wfuzz.readthedocs.io/en/latest/`. Accessed 2022-3-10.

[31] Xavier Mendez. Wfuzz - github repository, Accessed 2022-3-10. URL `https://github.com/xmendez/wfuzz`.

[32] Barton Miller, David Koski, Cjin Lee, Vivekananda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. Fuzz revisited: A re-examination of the reliability of unix utilities and services. Technical report, University of Wisconsin - Computer Sciences Department, 01 1998.

[33] Stefan Nagy, Anh Nguyen-Tuong, Jason D. Hiser, Jack W. Davidson, and Matthew Hicks. Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1683–1700. USENIX Association, August 2021. URL https://www.usenix.org/conference/usenixsecurity21/presentation/nagy.

[34] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, page 89–100. Association for Computing Machinery, 2007. doi: 10.1145/1250734.1250746. URL https://doi.org/10.1145/1250734.1250746.

[35] Information Sciences Institute University of Southern California. RFC 793 - transmission control protocol, September 1981. URL https://datatracker.ietf.org/doc/html/rfc793.

[36] The University of Wisconsin Madison. Fuzz testing of application reliability, Accessed 2022-3-8. URL https://pages.cs.wisc.edu/~bart/fuzz/.

[37] Joshua Pereyda. *boofuzz: Network Protocol Fuzzing for Humans*, October 2020. URL https://boofuzz.readthedocs.io/. Accessed 2022-3-10.

[38] Joshua Pereyda. Boofuzz - github repository, Accessed 2022-3-10. URL https://github.com/jtpereyda/boofuzz.

[39] 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, 2020. doi: 10.1109/ICST46399.2020.00062.

[40] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. AFLNet - github repository, Accessed 2022-3-10. URL https://github.com/aflnet/aflnet.

[41] J. Reschke R. Fielding. RFC 7230 - hypertext transfer protocol, June 2014. URL https://datatracker.ietf.org/doc/html/rfc7230.

[42] J. Reschke R. Fielding. RFC 7231 - hypertext transfer protocol - semantics and content, June 2014. URL https://datatracker.ietf.org/doc/html/rfc7231.

[43] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-Assisted feedback fuzzing for OS kernels. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 167–182, Vancouver, BC, August 2017. USENIX Association. URL https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schumilo.

[44] Tim Schürmann. Hiawatha webserver, Accessed 2022-4-6. URL https://www.hiawatha-webserver.org/.

[45] Yan Shoshitaishvili. Preeny - github repository, Accessed 2022-3-7. URL https://github.com/zardus/preeny.

[46] Prashast Srivastava and Mathias Payer. Gramatron: Effective grammar-aware fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, page 244–256. Association for Computing Machinery, 2021. doi: 10.1145/3460319.3464814. URL https://doi.org/10.1145/3460319.3464814.

[47] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Network and Distributed Systems Security (NDSS) Symposium*, 01 2016. doi: 10.14722/ndss.2016.23368.

[48] Igor Sysoev and Inc. Nginx. Nginx, Accessed 2022-3-11. URL https://nginx.org/en/.

[49] GCC Team. *Using the GNU Compiler Collection (GCC) - Gcov - a Test Coverage Program*, Accessed 2022-2-23. URL https://gcc.gnu.org/onlinedocs/gcc/Gcov.html#Gcov.

[50] The Clang Team. *Clang Compiler Users's Manual - UndefinedBehaviorSanitizer*, Accessed 2022-2-23. URL https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[51] The Clang Team. *Clang Compiler User's Manual - AddressSanitizer*, Accessed 2022-5-19. URL https://clang.llvm.org/docs/AddressSanitizer.html.

[52] The Clang Team. *Clang Compiler User's Manual - SanitizerCoverage*, Accessed 2022-5-19. URL https://clang.llvm.org/docs/SanitizerCoverage.html.

[53] The Clang Team. *Clang Compiler User's Manual - ThreadSanitizer*, Accessed 2022-5-19. URL https://clang.llvm.org/docs/ThreadSanitizer.html.

[54] LiteSpeed Technologies. OpenLiteSpeed Website, Accessed 2022-4-6. URL https://openlitespeed.org/.

[55] L. Torvalds and Git Community. Git - version control system, Accessed 2022-4-4. URL https://git-scm.com/.

[56] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM : A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 745–761, Baltimore, MD, August 2018. USENIX Association. URL https://www.usenix.org/conference/usenixsecurity18/presentation/yun.

[57] Michał Zalewski. Technical "whitepaper" for afl-fuzz, Accessed 2022-3-8. URL https://lcamtuf.coredump.cx/afl/technical_details.txt.

[58] Google Project Zero. WinAFL - github repository, Accessed 2022-4-1. URL https://github.com/googleprojectzero/winafl.