5-2012

A Framework for File Format Fuzzing with Genetic Algorithms

Roger Lee Seagle Jr.
University of Tennessee - Knoxville, rseagle@utk.edu

To the Graduate Council:

I am submitting herewith a dissertation written by Roger Lee Seagle Jr. entitled "A Framework for File Format Fuzzing with Genetic Algorithms." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

Michael D. Vose, Major Professor

We have read this dissertation and recommend its acceptance:

Itamar Arel, Bradley Vander Zanden, Sergey Gavrilets

Accepted for the Council:
Dixie L. Thompson

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

A Framework for File Format Fuzzing with Genetic Algorithms

A Thesis Presented for

The Doctor of Philosphy

Degree

The University of Tennessee, Knoxville

Roger Lee Seagle, Jr.

May 2012

Dedication

To my wife, Mindy and my son, Bodhi, with your constant love and inspiration, you put my dreams within reach.

Acknowledgements

Over the course of my academic career, I have had the distinct pleasure of being enlightened by a prestigious group of academics. From this group, several individuals have fostered an everlasting love for computer science and made a significant impact upon my career. For their infinite erudition, I owe a debt of gratitude.

First and foremost, I am grateful for my academic advisor, Dr. Michael D. Vose. His guidance, numerous reviews, and wealth of knowledge shaped my vision into fruition. He demonstrated a mastery of knowledge and ingenuity to which I can only strive to procure. I am also grateful for my committee, Dr. Itamar Arel, Dr. Tom Dunigan, Dr. Sergey Gavrilets, and Dr. Bradley Vander Zanden, for their advice and feedback which greatly refined my research. While my professors from my graduate institution have been a significant influence, I cannot omit professors from my undergraduate institution, Wake Forest University. They are responsible for planting the seed that culminated in this dissertation. I am thankful for Dr. David John for introducing me to genetic algorithms and Dr. Errin Fulp for igniting a passion for computer security.

However, none of this would have been possible without extensive support and guidance from my family. I am thankful for my father, Dr. Roger Seagle Sr., for instilling a life-long pursuit of learning and exemplifying the perseverance and determination needed to achieve my dreams. I am grateful for my mother, Janie Halley, whose nurturing spirt laid a foundation upon which my career has been built. To my wife's parents, Jackie and Lisa Shelton, I owe a special thanks for their enduring support throughout this process. I thank my step-parents for their love and compassion, my grandparents who continue to be role models and challenge me to be a better person, and my siblings for embodying the true spirit of life.

Last but not least, I am grateful for my friends and colleagues, Mark Eklund, Kenny Gilbert, Jon Gill, Jay Koehler, Steve Rich, and Mark Shirley, who have consistently introduced me to bleeding edge technologies, inspired me with their fervor and openly shared their creative forces. It is their provocation that makes me aspire to be a better computer scientist.

Abstract

Secure software, meaning software free from vulnerabilities, is desirable in today's marketplace. Consumers are beginning to value a product's security posture as well as its functionality. Software development companies are recognizing this trend, and they are factoring security into their entire software development lifecycle. Secure development practices like threat modeling, static analysis, safe programming libraries, run-time protections, and software verification are being mandated during product development. Mandating these practices improves a product's security posture before customer delivery, and these practices increase the difficulty of discovering and exploiting vulnerabilities.

Since the 1980's, security researchers have uncovered software defects by fuzz testing an application. In fuzz testing's infancy, randomly generated data could discover multiple defects quickly. However, as software matures and software development companies integrate secure development practices into their development life cycles, fuzzers must apply more sophisticated techniques in order to retain their ability to uncover defects. Fuzz testing must evolve, and fuzz testing practitioners must devise new algorithms to exercise an application in unexpected ways.

This dissertation's objective is to create a proof-of-concept genetic algorithm fuzz testing framework to exercise an application's file format parsing routines. The framework includes multiple genetic algorithm variations, provides a configuration scheme, and correlates data gathered from static and dynamic analysis to guide negative test case evolution. Experiments conducted for this dissertation illustrate the effectiveness of a genetic algorithm fuzzer in comparison to standard fuzz testing tools. The experiments showcase a genetic algorithm fuzzer's ability to discover multiple unique defects within a limited number of negative test cases. These experiments also highlight an application's increased execution time when fuzzing with a genetic algorithm. To combat increased execution time, a distributed architecture is implemented and additional experiments demonstrate a decrease in execution time comparable to standard fuzz testing tools. A final set of experiments provide guidance on fitness function selection with a CHC genetic algorithm fuzzer with different population size configurations.

Contents

List of Figures

Chapter 1

Introduction

Software testing is an integral component of a software development life cycle. Many different software development processes, from older constructs such as the Spiral Model [Boehm 1986] to modern practices like Agile Development [Beck 1999], mandate thorough testing. Oftentimes, these models differentiate themselves by prescribing a unique testing methodology or testing interval. While these variances can help classify a development process, the goal behind prescribing software testing remains consistent. This goal is to improve software quality and resiliency through structured, rigorous verification and validation.

Verification and validation form a basis for software quality control and testing theory. A tester assessing software in these terms strives to assure a developer that software is built without design or implementation errors (verification) and a customer that software fulfills its intended purpose (validation). More formally, verification occurs throughout a development process and determines an implementation's correctness. Tasks associated with verification assist a tester in reducing occurrences of defects. Validation determines if the software was developed according to customer requirements and specification. Requirements are desired features and functions, and a specification defines the software's behavior in formalized documentation based upon the requirements. Validation assures a customer that software developers implemented software to meet the customer's requirements. Together, verification and validation improve software quality and resiliency.

A vast array of concepts, tools, and techniques comprise software testing and facilitate verifying and validating software. These range from evaluating software's ability to consistently operate in different environments, known as compatibility testing, to assuring correct operation of specified sections of code, referred to as unit testing. Each concept, tool, or technique can have positive and negative characteristics and may even focus on eliminating a particular flaw. For instance,

stress testing evaluates the ability of a program to operate under heavy load. It measures resiliency under undue stress, but, contrary to unit testing, it fails to verify the correctness of implemented functionality.

Fuzz testing, or more commonly referred to as fuzzing, is a popular software testing technique supporting software verification and to a lesser degree validation. In most instances, fuzzing discovers implementation flaws, yet, in other cases, it can discover undesirable functionality not defined by customer requirements. This broad range of defect discovery highlights the difficulty of arranging testing techniques into discrete categories. Nevertheless whether fuzz testing, stress testing, or unit testing, an understanding of fundamental software testing terminology is desirable.

One of the more important distinctions in terminology is the dichotomy between structural and functional testing. Structural testing is the practice of systematically reviewing the requirements and specification of a software program to determine if any mistakes or omissions are present. It analyzes requirements along with the specification to clarify vagueness that may lead to undesirable functionality or costly errors. An example of structural testing is the Sequence-Based System Specification. This technique improves software quality by clarifying requirements through a systematic enumeration of possible inputs and outputs to a software system [Prowell and Poore 2003]. This thorough enumeration prevents developers from misinterpreting a requirement and uncovers differences between the requirements, specification, and actual implementation. The usage of the Sequence-Based Specification combined with statistical testing reduces the cost of software testing and improves software quality [Prowell 1998].

Functional testing abandons in-depth analysis of requirements and specifications to focus on behavioral analysis of an implementation. Crafted test cases exercise portions of software to gauge whether it behaves as expected, erroneously, throws an exception, or crashes. The composition and development of a test case directly impacts the ability of functional testing to find defects. Fuzz testing is generally classified as a version of functional testing even though it can uncover gaps between the requirements and implementation [Takanen et al. 2008].

Another important distinction in software testing terminology, applicable to the classification of fuzz testing, is the difference between white, gray, and black-box testing. The delineation between these three terms depends upon the amount of information available about a program under test. For instance, if a program's source code is available to the individual testing the software, then the testing is referred to as white-box testing.

With black-box testing, test cases are crafted without information about the application's architecture or implementation. The only information gathered from the program under test is the response to a stimulus, hence, the term black-box. Many forms of functional testing are deemed black-box testing techniques. In its purest form, fuzzing is a black-box testing technique.

Gray-box testing mixes aspects of black-box and white-box testing. In this testing form, partial information about a program is available to guide test case generation. An example of partial information is a referral to a program's disassembly for assigning code coverage metrics to a test case. The disassembly provides the tester with information about code structure, flow, and behavior, but it obfuscates higher-level details. Black box testing may consult other information such as a program's Application Programming Interface (API) or architecture documentation. However, the software tester does not have a complete listing of the program's source code.

The overarching goal of this research is to improve gray-box functional testing by extending current fuzz testing techniques. Much research has been conducted in this area thereby improving software quality. While previous research has made great strides in fuzz testing, the algorithms developed can be built upon and improved. In order to understand where improvements are possible, the next section presents the current problem with fuzz testing algorithms.

1.1 Problem Statement

Software testing, as shown by the previous section and documented further in [Beizer 1990], is an important, complex field of Computer Science. It encompasses a wide array of concepts and tools to help improve software quality. Whether through rigorous verification by software testers or a comprehensive validation process whereby requirements are traced and matched to implemented features, software requires intense scrutiny to improve its robustness, resiliency, and stability.

Many different testing techniques exist to assess and improve software based on these characteristics. One well-known method of testing is to enumerate a program's input vectors and provide each vector with a set of negative test cases. An input vector is a method a program uses to collect data, such as file IO or network connections, and a negative test case refers to input designed to test a program's ability to handle erroneous data. Negative test cases contain common data type boundary condition values like the maximum value of an integer, a nefariously long string, invalid character sequences, or even a randomly generated byte string. By enumerating a program's input vectors with negative test cases, a software tester can assess whether a program can continue operating as expected when presented with erroneous data. If a program processes these values without crashing or exhibiting other undesirable behavior, then a program correctly handles a negative test case and passes the test. Otherwise, a negative test case pinpoints a software defect, and the program fails the test.

The generation and delivery of negative test cases along with monitoring, logging, and assessing a program's ability to process a negative test case is commonly referred to as fuzz testing. This type of testing uses a tool, known as a fuzzer or fuzzing framework, to automate negative test case

creation, delivery, program monitoring, and logging. An assessment of program behavior can also be included in the fuzzer's functionality, yet this assessment can also be handled by accompanying tools.

A fuzzer's main challenge is to efficiently generate negative test cases from the possible input space which are likely to elicit program errors. An input space is all the possible byte sequences capable of being processed by a program. Since this collection of permutations is typically quite large, it is infeasible for a fuzzer to enumerate an entire input space in a reasonable time period. To combat this challenge, over the past ten years, researchers have started to apply search algorithms for finding and generating good negative test cases. Within the past five years Sparks et al. [2007] and DeMott et al. [2007] introduced genetic algorithms as a viable solution to this problem.

The combination of a genetic algorithm to explore an input space for negative test cases paired with a fuzzer to test a program appears promising. However, the genetic algorithms combined with fuzzers eliminated the ability to define custom fitness functions which, in turn, can constrict the evolutionary process. These genetic algorithms currently focus upon evolving negative test cases to locate and exercise unsafe library functions (strcpy(), strncpy(), strcat(), etc.) or improve code coverage by rewarding a negative test case that exercises less frequently executed code. This focus can constrain evolution and may impede a fuzzer's ability to adequately locate other software defects. The effectiveness and flexibility of genetic algorithm fuzzing can be improved by broadening the current focus to incorporate additional information from static and dynamic program analysis. This research develops a gray-box proof-of-concept genetic algorithm fuzzing framework aspiring to reveal software defects such as integer overflows, heap overflows, stack overflows, double frees, signed comparisons, and off-by-one errors.

1.2 Approach

The proof-of-concept fuzzing framework developed through this research introduces a novel approach to negative test case generation by combining concepts from reverse engineering and search algorithms. These concepts are:

1. *Static Analysis* - A technique for reviewing program behavior without the benefit of code execution. The analysis can be performed either directly on the program's source code, or, in the absence of source code, a disassembler can translate a program's machine code to assembly language before analysis. The analysis can then proceed manually, by predefined algorithms, or through scriptable interfaces. Examples of static analysis tools are Coverity Prevent [Coverity 2010], Klocwork Insight [Klocwork 2010], and Fortify 360 [Fortify Software 2010]. This research

assumes a software tester is evaluating a closed source application. All static analysis occurs on assembly code generated by a disassembler.

2. *Dynamic Analysis* - A technique for analyzing program behavior by monitoring, recording, or altering program execution. One strategy of dynamic analysis is to emulate a program to determine its behavior. Program emulation isolates a program to a controllable sandbox where instrumentation can then be applied. Tools categorized as dynamic analysis emulation tools include DynamoRIO [Bruening 2004], Pin [keung Luk et al. 2005], and Valgrind [Nethercote 2004].

3. *Genetic Algorithms* - An evolutionary algorithm mimicking darwinian evolution to explore search spaces of optimization problems. These algorithms evolve candidate solutions to a problem by applying genetic operators, such as selection, crossover, and mutation, to a population of previous candidate solutions. Several other flavors of evolutionary algorithms exist and include evolutionary programming [Fogel et al. 1966], evolutionary strategies [Rechenberg 1971], and genetic programming [Koza 1990].

By combining these concepts, several of the fuzzers described in Chapter 2 are unified. This unification extends the canon of fuzz testing methodology by increasing the number of concepts incorporated into fuzz testing.

This research specifically improves upon genetic algorithm fuzzing by addressing limitations with fitness function metrics. A fitness function and selection algorithm heavily impact a genetic algorithm's ability to find an optimal solution to a search or optimization problem. Poor fitness functions may incompletely characterize a search space and assign incorrect weights to population members. These weights may then cause a selection algorithm to favor suboptimal solutions leading to a divergence from an optimal solution or a convergence to an inadequate solution. Fitness functions and selection algorithms must be evaluated to guard against these undesirable situations [Sivanandam and Deepa 2007a].

This research extends the metrics available for specifying fitness functions when fuzzing. The extension allows software testers to tailor fitness functions by composing mathematical equations based on data collected from static and dynamic analysis. Tailoring a fitness function on a case by case basis improves the flexibility of genetic algorithm fuzzers. Whereas a fitness function rewarding code coverage could equate to more resilient code for some programs, it might for other programs fail to efficiently evolve a useful collection of negative test cases. By extending available metrics and exposing a set of variables to compose a mathematical equation for a fitness function, this research significantly advances genetic algorithm fuzz testing.

1.3 Contributions

The main contribution of this research is the development of a proof-of-concept fuzzing framework with multiple genetic algorithms and configurable fitness functions based on static and dynamic analysis. This notion of a configurable fitness function is novel in the domain of genetic algorithm fuzzing; previous research efforts present functions fine-tuned to evolve towards known unsafe C APIs or maximize code coverage. These fine-tuned functions were intended to unearth common programmer errors, yet the fuzzers built upon these fitness functions lock users into a particular evolutionary model. The state of the art in fuzz testing is enriched by unlocking fitness function composition and applying other genetic algorithms.

Also, in support of this main contribution, this research enhances fuzz testing by:

1. *Integrating emulation monitoring into genetic algorithm fuzzing.* Molnar and Wagner [2007] and Campana [2009] demonstrate the usefulness of combining symbolic execution and constraint solving with fuzzing. Their research illustrates the ability to find integer conversion errors and target fuzzing by tracing tainted data. This research combines the proven concept of program emulation with genetic algorithms to monitor program execution and guide negative test case evolution.

2. *Establishing a set of variables about program attributes.* A variable is a symbol created to abstractly represent information which could change. In this case, a variables coalesces data from static and dynamic analysis to describe a computer program's observed behavior. Static analysis extracts attributes about assembly code structure, and dynamic analysis determines the actual code paths traversed during an instantiation of a program with a test case. When this data is combined, the measurements paint a more complete picture of program behavior and resource utilization.

3. *Presenting a configurable genetic algorithm fuzzer.* Applications of genetic algorithms to fuzz testing are incompletely described or do not permit users to alter parameters influencing evolution. For example, the source code for the genetic algorithm of Sparks et al. [2007] is unavailable, and the fitness function of DeMott et al. [2007] is fixed. This research provides a complete genetic algorithm fuzzing framework allowing the user to tailor genetic algorithm parameters.

4. *Implementing an ad hoc architecture for distributed, co-operative fuzzing.* The ability of fuzzing algorithms to execute test cases in a timely manner diminishes as the complexity of the algorithm increases. This observation is the archetypical struggle between speed and complexity. Fuzzer effectiveness, in part, relies upon an algorithm's ability to balance this

struggle in order to generate and execute large numbers of test cases. The absence of this balance inhibits an algorithm's performance. This research implements a distributed model of fuzzing to combat algorithm complexity. The architecture builds upon open source software which permits any individual the opportunity to deploy an ad hoc open source co-operative fuzzing environment. The framework extends previous research on distributed file fuzzing by demonstrating an open source, ad hoc clustering model [Conger et al. 2010].

5. *Providing a comparative analysis of genetic algorithm fuzzing against other common fuzzing algorithms.* The true measure of fuzzer effectiveness is its ability to discover software errors. A case study is presented within this dissertation to articulate the pros and cons of genetic algorithm fuzzing with regards to other available fuzzers. The case study gauges effectiveness by systematically testing a commercially released software application with the implemented genetic algorithm fuzzing framework and other common fuzzing algorithms.

6. *Comparing and contrasting multiple genetic algorithm fitness functions based on program attributes.* The exportation of static and dynamic analysis attributes to users for inclusion into a fitness function is not wholly sufficient to increase the ability of a genetic algorithm fuzzer to find software errors. Extra information may cause performance degradation if users are given options but no direction concerning their effective use. A lack of direction may prohibit users from determining any appropriate fitness measures in a timely manner. This research addresses that issue by exploring several candidate fitness functions to provide users with a baseline of knowledge about fitness function composition.

1.4 Scope

The problem outlined by this dissertation is addressed through the design and development of an extensible, configurable proof-of-concept gray-box genetic algorithm fuzzing framework. The framework focuses on validating the robustness of a program's file parsing routines. It disregards testing other input vectors such as network communication, environment variables, or command line parsing. While it is important to harden these interfaces, file parsing presents an easily accessible input vector to experiment with new fuzzing algorithms.

Also, file parsing functions and libraries notoriously contain defects targeted by malicious hackers. An example of a common defect leveraged by malicious hackers is a function blindly trusting file content specifying data length values. Defects, such as these, empower knowledgable individuals with the ability to gain arbitrary control of a victim's computer by specifying incorrect data length values and overwriting important sections of memory with malicious data. Since a malevolent individual

can create arbitrary files and distribute them freely on the Internet. A program processing files should treat file content as an untrusted data source. File content should always be validated before processing. Sutton and Greene [2005] and Miller [2010] demonstrate the relevance and fruitfulness of this input vector, even with software released today.

The fuzzing framework developed in this dissertation only targets applications running on the Mac OS X operating system. The Mac OS X operating system is quickly gaining market share against companies like Microsoft. For years, security researchers have chosen to develop fuzzers for Windows or Linux due to the impact on consumers or ease of implementation. Mac OS X presents fertile territory for fuzzing research. The concepts developed by this research, while implemented in a Mac OS X environment, are portable to other operating systems.

The structure of this dissertation is as follows. Chapter 2 presents the basic anatomy of a fuzzer and a literature review of common fuzzing algorithms. Chapter 3 discusses the requisite pre-processing for a genetic algorithm fuzzing framework, namely binary analysis. Chapter 4 presents the Mamba fuzzing framework and examines the inclusion of program emulation into the framework. Chapter 5 explains the addition of a genetic algorithm to the Mamba fuzzing framework, and it introduces a set of variables for defining fitness functions representing program attributes extracted during binary analysis and observed during program execution. Chapter 6 defines Mamba's distributed, co-operative file fuzzing architecture, and illustrates improvements in execution time for a genetic algorithm by using the distributed architecture. Chapter 7 presents results from experiments testing multiple genetic algorithm fitness functions and population sizes. Chapter 8 concludes the dissertation with overarching observations and proposals for future research.

Chapter 2

Literature Review

Many companies and individuals today utilize open source or proprietary fuzzers during software testing to ensure program robustness. The term fuzzing means presenting a program's input vectors with random or anomalous data to test a parser's reliability. Each anomalous input (negative test case) tries to locate programming errors that could cause memory corruption or even aid in the execution of maliciously injected code. Microsoft's distributed fuzzing framework, recently publicized for finding 1,800 defects in Microsoft Office 2010, is a prime example of the increased reliance on fuzz testing [Conger et al. 2010].

A fuzzer's goal, when used by a company, is to eliminate costly programmer errors before shipping software to consumers, or proactively find software defects before customers, vulnerability researchers, or malevolent entities encounter them. If a defect is found after shipment by a vulnerability researcher or a malevolent entity, then the ramifications to the software company are significantly more severe than discovery during internal testing. Examples of possible ramifications are public scrutiny of software design, revenue shortfalls, or low customer adoption. While fuzzers can reduce the occurrence of externally found defects, they alone are not a software testing panacea. A well defined software test plan incorporating periodic code reviews and unit testing coupled with fuzzing is superior to only fuzz testing [Chess et al. 2010].

Fuzzing genealogy and even the term "fuzz" originates from a researcher at the University of Wisconsin, Madison named Barton Miller. In the mid to late eighties, Professor Miller observed that thunderstorms caused line noise in his modem connection to remote computers. This line noise periodically injected junk characters into the remote sessions causing interference with program operation. Eventually, programs crashed due to these junk characters, and, Professor Miller coined the term "fuzz" to describe this behavior [Neystadt 2008].

Over the next few years, Professor Miller's curiosity with random data injection grew. In the Fall of 1998, he designed an operating systems class project based upon fuzzing. The assignment challenged students to write a program for testing the reliability of standard Unix utilities' options parsing routines. These programs, now referred to as a fuzzer or fuzzing framework, generated random data, delivered the data to the parsing routines of a program under test, and monitored that program for faults [Millller 1988].

Most of the student's fuzzers were lackluster, however, the results from one student's submission were astounding. These results showed that even though Unix utilities were regarded as stable, several of their options parsing routines lacked proper command line argument sanitization. When these parsing routines received anomalous data from command line arguments, the absence of parameter sanitization and bounds checking caused segmentation violations and other exceptions. These observations were so groundbreaking that they spawned the development of a software testing group at the University of Madison, Wisconsin. This research group later released a report in 1990 detailing the defects observed from fuzzing Unix utilities [Miller et al. 1990], and this report sparked the beginning of formal research in fuzz testing by academics and security researchers.

Fuzz testing methodology has matured since its initial discovery by Professor Miller. A fuzzer is now expected to adhere to a minimum set of requirements. Most notably, Sutton et al. [2007] advocate for a fuzzer to properly document test cases, guarantee reproducibility, effectively monitor a program under test, provide informative metrics, and properly detect program exceptions. These requirements are derived directly from their experiences with building real-world fuzzing frameworks and through observations of other fuzzer implementations. These requirements describe an acceptable baseline for fuzzer development, and each requirement is vital for creating a usable and effective testing tool.

Test case documentation informs a researcher of a negative test case's storage location, time at which a program ran the test case, and the result of executing a program with the test case. This documentation guarantees that the observed behavior can be reliably reproduced. If a negative test case is not properly documented and stored, then a crash caused by it is incapable of being reproduced or analyzed. Improper documentation and storage can also cause a higher false positive rate. In this context, a false positive indicates a negative test case causes a program exception when in actuality program execution terminates normally.

Process monitoring tracks a program's state during the execution of a negative test case. Information collected from the monitoring serves as historical data about program behavior. The historical data can then be aggregated to generate useful metrics, such as code coverage. Various forms of code coverage metrics exist like branch coverage. This metric calculates the number of branches executed by a test case out of the total number of branches within a program [Beizer

1990]. For completeness, a branch is a program decision point where execution can proceed in more than one alternative. Accurate and relevant metrics, like branch coverage, are helpful in determining the amount of code tested by a fuzzer.

Error detection, also known as exception monitoring or health monitoring, determines a program's reaction to a test case by examining process state after execution ceases. A monitor evaluates process state for abnormal exit conditions like segmentation violations, bus errors, and arithmetic exceptions among many others. The fidelity of an exception monitor impacts the rate of false positives as well as false negatives. A false negative states a test case causes a program to exit normally when, in reality, the test case causes an exception. Issues about exception monitoring fidelity are important to consider while developing or evaluating a fuzzer.

Takanen et al. [2008] extend upon Sutton's requirements by encouraging the addition of exception analysis. Exception analysis combines information gathered during process and exception monitoring to determine the cause and impact of an observed exception. The effort required for this analysis is dependent upon the level of detail obtained from process and exception monitoring. Exceptions, also referred to as program faults, have underlying defects ranging in severity from benign Denial Of Service (DoS) to catastrophic remotely exploitable memory corruption. Defects resulting in DoS are less severe than remotely exploitable defects, because it is infeasible to run arbitrary malicious code. With remotely exploitable defects, an attacker may be capable of exploiting the defect and assuming full control of the running process. To determine a defect's cause and severity, exception analysis can be conducted by rerunning a test case with a debugger connected to the program under test, inserting special debugging shared libraries (e.g. libgmalloc), or by running third party utilities to parse and interpret error logs (e.g. !exploitable or crashwrangler). A determination of cause and severity derived from exception analysis informs a developer of a defect's impact to a program's customers.

While the aforementioned requirements vividly describe fuzzer characteristics, they do not provide a taxonomy of fuzzers. Figure 2.1 illustrates a rough taxonomy with three different classifications. Each classification contains a set of required and optional components correlating directly with one or more of the requirements stated by Sutton et al. [2007] and Takanen et al. [2008]. These essential components are a set of attack heuristics, an executor, a monitor, a logger, and a classifier. A solid border around a component signifies a required component and a dashed border signifies an optional component.

Fuzzer classification primarily relies on a dissection of a fuzzer's algorithm for generating negative test cases. This algorithm determines the method of injecting anomalous data from the set of attack heuristics into a negative test case. Different classifications have adopted different methods of injecting this data and their methodology can uniquely determine whether a fuzzer can be classified

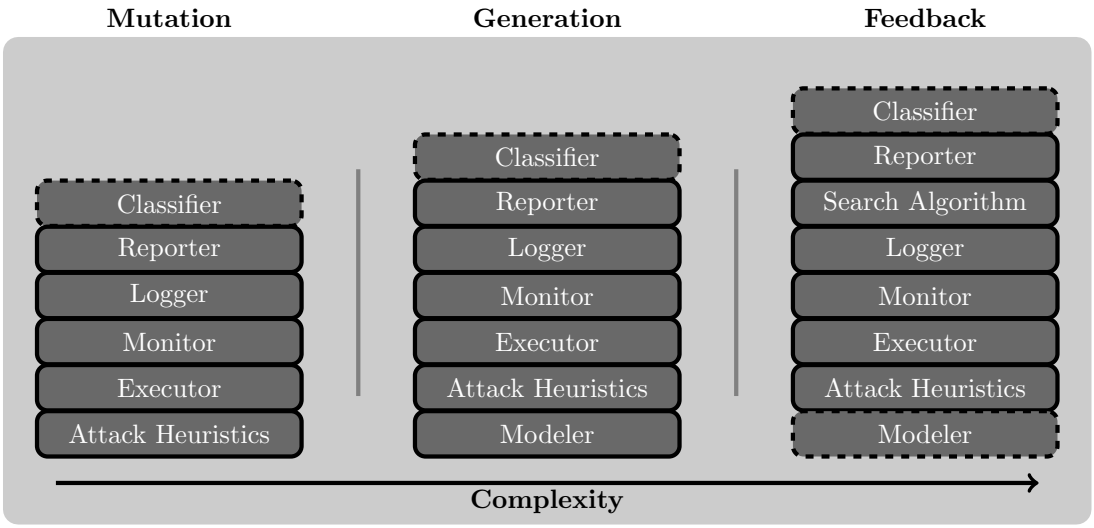


Figure 2.1: Fuzzer Classifications

as a mutation, generation, or feedback fuzzer. While different fuzzer requirements have rapidly surfaced, evolution of different fuzzer classifications has been slower. Mutation fuzzers were the only classification until the turn of the 21st century when generation fuzzers were introduced. Only recently, over the past five years, feedback fuzzers have debuted and increased in prevalence.

Attack heuristics define the set of anomalous data to be inserted by a fuzzer into a test case. More formally, attack heuristics are techniques proven to find defects in applications by past experiences [Takanen et al. 2008]. These techniques could be random, such as sampling from /dev/urandom, inserting historically known troublesome values, or any combination thereof. An instance of known troublesome values are boundary conditions of basic types. For instance, the maximum value of an 8-bit unsigned char or a 16-bit unsigned short may be members of an attack heuristic set. These values, 255 and 65,535, are in the attack heuristics of several fuzzers and may cause programs to crash [Amini and Portnoy 2007, Eddington 2006] . A fuzzer, either randomly or systematically, selects values from this set and inserts the selected value into a negative test case. The quality and number of attack heuristics can impact the ability of a fuzzer to find defects.

An executor is the mechanism a fuzzer uses to run a program under test and deliver a generated negative test case. Executor design depends on several factors such as the programs intended to be tested, input vectors, and operating system. An input vector is the avenue which data is presented to an application for processing. Two common input vectors are network traffic and files.

Executor design is dependent on multiple factors, because different programs and operating systems handle input vectors in different manners. For instance, if a program under test accepts command line arguments to select file input and the designated input vector to fuzz is file parsing,

then an executor only requires running a program with the correct command line arguments and negative test case. In another instance, a program may not accept command line arguments to specify file input. The lack of a command line argument for file input increases executor complexity, since the executor must present the file through a different method. One way this can be achieved on the Mac OS X operating system is through the Applescript language. This scripting language controls scriptable applications with actions like file opening, program closing, and file saving among many others. A well designed executor accounts for differences in program operation and presents robust cross-platform interaction and control [Apple 2008].

A monitor or health monitor watches the program under test in an effort to determine the outcome from running a negative test case. As with an executor, the target application and, to a larger degree, the operating system determines the method of monitoring. Several different methods exist to reliably monitor and detect application faults. A program under test can be monitored by starting the process with an attached debugger. The debugger automatically catches faults generated by an application, and, the debugger provides detailed information about a fault. Monitoring with a debugger can also be achieved with scriptable interfaces like the popular Python library pydbg [Amini 2006] or the newly released Ruby library Ragweed [Monti et al. 2009]. Scriptable interfaces are convenient, but unfortunately, they can be platform dependent.

Another option for monitoring is to detect the operating system writing to an error log. One example, on Mac OS X, is the directory ~/Library/Logs/CrashReporter. If a monitor watches this directory for changes, then it can detect program faults. However, since directory watching does not monitor program execution, the information collected by this method is not as detailed as monitoring with a debugger. Error directory or file watching is also platform dependent in terms of the files to watch and the different watching interfaces. This platform dependency is evident when comparing Linux and Mac OS X. For Linux, directories and files are capable of being watched with the inotify library, and for Mac OS X, a program can register to receive notifications of directory changes through the FSEvents API.

Another, more heavy-weight monitoring option, is to emulate the program under test with a Dynamic Binary Instrumentation Framework (DBI). These frameworks translate native instructions into an intermediate representation thereby providing hooks for developers to interact directly with the translation during execution. The translated instructions are instrumented based on the programmer hooks, and then the intermediate representation is executed. Since the DBI translates every instruction and executes the translation, every detail about execution can be gathered. Faults are capable of being detected in real-time and detailed information about execution can be logged.

A logger records fuzzer progress with a log of fuzzer events which can be audited later. The log contains details such as the time each negative test case is generated, location of a negative test

case, and any fuzzer specific debugging information. A logger can also catalog fuzzer configurations and start and stop times for performance comparisons. Logs created by the logger aid a tester in fuzzer development and auditing fuzzer results. Logs are the primary resource for debugging and historical data.

Reporters summarize pertinent information on demand about a fuzzer run and notify a tester of important events. Summary information includes the number of test cases executed, current runtime, and the current list of test cases causing faults. A notable event for the reporter is a program generating an exception from a test case. Instant program crash notification provides instant feedback about the fuzzer's attack heuristics and algorithm. Loggers and reporters are similar, yet a reporter is distinguishable by its on demand summarization and instant notification. By contrast, a logger records more detailed information for later review after a fuzzer concludes testing.

The last core component of a fuzzer is optional and known as a classifier. Classifiers coalesce information collected by a monitor and logged by a logger. Once coalesced, the classifier determines cause and severity of program exceptions. This determination describes the impact of a negative test case on a program under test.

Classifiers are mostly relegated to post-processing, and these programs are written independently by major software vendors. These separate analysis programs run on the system crash logs after all the negative test cases have been executed or instrument the program under test and rerun the suspect negative test case. They attempt to sort crashes in an effort to determine their uniqueness. Two of the prominent classifiers are Mac OS X's crashwrangler and Microsoft's !exploitable. These classifiers, while generally sufficient for categorizing the types of faults found by a fuzzing run, are not always capable of determining cause or severity. Johnson and Miller [2010] recently discussed the known limitations of these analysis toolkits. Due to these limitations, other manual techniques are often required for crash analysis and classification.

Fuzzers implementing only the core components are classifiable as mutation fuzzers. Mutation fuzzing, synonymous with dump fuzzing, is the earliest classification developed by Barton Miller. These fuzzers generate negative test cases by mutating sample inputs with attack heuristics. For example, a valid image file could be altered by randomly selecting and flipping bytes. The fuzzing algorithm, in this example, is random selection and the attack heuristic is byte manipulation. A mutation fuzzer may also randomly select values of a formal attack heuristic set and insert those values directly into a sample valid file.

Mutation fuzzers are easy to implement and adapt well to testing arbitrary software. These qualities are a direct result from their simplistic design. However, mutation fuzzers lack the sophistication required to generate valid targeted tests. Some test cases may require updating a

checksum field based on an attack heuristic inserted in order to exercise different code sections. Mutation fuzzers lack this capability.

One of the most prominent mutation file fuzzers is Mangle written by Ilja van Sprundel. Released around 2005, Mangle targets the header section of input files. This section stores metadata pertaining to a file's content such as length information or tables of pointers to other sections within the file. By altering this information, a file parser may incorrectly parse another file section or even blindly trust incorrect length values [van Sprundel 2005].

van Sprundel's [2005] algorithm is elementary with the implementation only spanning around sixty lines of C code. Based upon pseudo-random numbers, it fuzzes between 0% to 10% of a file's header. The algorithm starts by requesting a random number from /dev/urandom. This number is then used as a seed for the standard C library pseudo-random number generator. Once seeded, a call to the rand() function determines the number of bytes to alter within the header bounded between 0% and 10%. Subsequent calls to rand() select the individual bytes to flip. These byte flips are achieved by flipping higher order bits of the selected bytes more often. Flipping the higher order bits targets integer overflows and signage errors.

While this algorithm is steeped in pseudo-randomness, it is responsible for uncovering many serious defects. It has found problems in Quicktime Player, Preview, and OpenOffice among many other programs. It does however fail to implement most of the requirements attributed to a fuzzing framework and some of the core fuzzing components. Most notably, it does not perform test case documentation or logging, process monitoring, or exception analysis. A shell script must be written wrapping the fuzzer to account for these missing features. Figure B.1, written by this author and found in the Appendix B with all other code examples, is an example wrapper bash script.

Sutton and Greene [2005] advanced mutation file fuzzing by introducing the notSPIKEfile fuzzer at the Blackhat security conference. This fuzzer is part of a duo of fuzzers based upon a general purpose network protocol fuzzer named SPIKE [Aitel 2002b]. notSPIKEfile implements most core components associated with a mutation fuzzer. It contains two sets of attack heuristics. One set of heuristics targets arbitrary binary values and another set targets troublesome strings. The binary heuristics test for integer overflows, while the other set scrutinizes string parsing. A couple of examples of troublesome strings are inordinately long strings or long strings with format specifiers. notSPIKEfile's code refers to these sets as the blob list and the string list. Other attack heuristics can be added to notSPIKEfile by inserting desired values into either set.

The algorithm behind this fuzzer, like Mangle and all other mutation fuzzers, requires a valid test case file as a baseline. For example, in the case of testing a Portable Document Format (PDF) reader, notSPIKEfile requires a valid PDF file. Once a tester provides a baseline test case, the fuzzer chooses one of the attack heuristic sets and substitutes every value in the set into the baseline

file byte by byte. Each substitution causes the program under test to be executed with the newly generated test case. Since each substitution generates a new negative test case, the number of test cases generated by this algorithm is the attack heuristic set's cardinality times the number of bytes in the baseline test case.

An interesting implementation detail of notSPIKEfile is its process monitoring and exception handling. Sutton and Greene [2005] chose the ptrace() system call for handling these tasks. The fuzzer forks the program under test, attaches to the forked process with ptrace(), and passively monitors execution for interesting signals. If a signal occurs, then the fuzzer determines if it is one of SIGILL, SIGBUS, SIGSEGV, or SIGSYS. These signals signify interesting deviations from normal execution, and, in the event of these signals being sent to the process, notSPIKEfile dumps the processor state of the target program for further analysis. If the observed signal is not one of the aforementioned, then the fuzzer passes the signal directly to the target process. This fuzzer is an improvement over Mangle due to the inclusion of process monitoring, fault detection, and logging.

The next category of fuzzing, generation or intelligent fuzzing, increases complexity by adding a modeling component to the set of core components. A modeler is an API for specifying a test case's format. Modeling APIs vary widely and are dependent upon a generation fuzzer's implementation. One fuzzer may define C functions to represent different formats, and another fuzzer may define a modeling language with an expressive markup language like Extensible Markup Language (XML).

Modelers increase the initial setup time required for starting a fuzzing job. Setup time is increased, because a tester must evaluate a format specification prior to testing. A format specification evaluation translates a specification into a fuzzer specific API and marks the fields desired for substitution. Researching format specifications may prove to be time consuming. If a format or protocol is proprietary, then reverse engineering the specification may preclude the ability to precisely define a model in a timely manner.

Furthermore, a generation fuzzer's ability to elicit defects is based partially upon model accuracy and attack heuristics. If the attack heuristics do not include sufficient anomalies or appropriate portions of the model are not marked for substitution, then some defects may not be found. Attack heuristic quality is a concern of all fuzzers. While increased setup time, model accuracy, and attack heuristics are concerns with generation fuzzing, a precise protocol model targets specific code sections. It aids a fuzzer in uncovering defects hidden in complex parsing logic.

Sutton and Greene [2005] presented a robust intelligent file fuzzer at the Blackhat security conference. The fuzzer, SPIKEfile, compliments the mutation fuzzer, notSPIKEfile, which they presented at the same conference. SPIKEfile encompasses most of the functionality of its relative, yet it extends functionality by adding block-based modeling. As detailed by Aitel [2002a] in his seminal paper on network fuzzing, block-based modeling examines the interaction between protocol

layers. Such modeling exemplifies the connection between layered protocol header size fields and the data encapsulated by the headers. With this interconnection in mind, Aitel [2002a] presented a C style scripting language for modeling network traffic by breaking it into blocks with length and data fields. Sutton and Greene [2005] applied this concept to file fuzzing with SPIKEfile.

SPIKEfile requires a baseline test case like notSPIKEfile, but it also requires a model file. Figure B.2 illustrates an example model of a Tagged Image File Format (TIFF) model based on the TIFF specification. As shown, the scripting language defined by Aitel [2002a] conforms to the C standard. All statements in the scripting language are C function calls with literal string arguments enclosed by quotes. Statements are terminated by semicolons and comments can be added throughout a model by starting the comment line with two forward slashes (//).

The data model defined in Figure B.2 illustrates a few of the multiple features of notSPIKEfile's modeling language. It starts by defining a fundamental modeling structure, a block. Blocks are composed of one or more data fields and an optional length field automatically calculated based upon the data field values contained within the block. The s_block_start() function call denotes the beginning of a block, and it is ended with the s_block_end() function call. Start and end functions require matching string parameters to designate the matching calls. If these parameters do not match, then an error occurs during model compilation.

Between block start and end calls reside example data. Figure B.2 demonstrates a block with multiple s_binary() function calls. This function requires a hex string argument, and it places that value directly in the generated test case. The location of the value within the generated test case is dictated by the function call's location within the model. Values defined with the s_binary() function are not marked for substitution. To mark a value for substitution, a function ending in the variable keyword must be used. The function s_string_variable() is an example function marking a string for substitution.

SPIKEfile's attack heuristics are split into two sets. A set for troublesome string values and a set for troublesome integer values. These sets are analogous to notSPIKEfile's attack heuristics. The fuzzing algorithm walks through the data model substituting appropriate attack heuristics into the marked variable. String variables are replaced with a value from the string attack heuristics, and integer variables are replaced with values from the integer attack heuristics. Since the algorithm walks through the entire file substituting marked variables, the number of negative test cases generated is equal to the summation of the number of variables marked for substitution times the cardinality of each variable's corresponding set of attack heuristics.

Eddington [2008] developed the most recognizable generation fuzzing framework around 2006. This fuzzer, Peach, while labeled as a generation fuzzer, is capable of operating as a mutation or generation fuzzer. It contains all of the core components attributed to a generation fuzzer along with

other features like distributed fuzzing and tools for baseline test case selection. Peach's portability, flexibility, logging, and integration with multiple debugging tools make it a robust framework.

Peach is configured by an XML document based on a published schema. The XML configuration defines a data model, state model, test configuration, and fault monitoring for a fuzzing run. The fuzzer also includes a Graphical User Interface (GUI) to simplify configuration absolving testers from learning the XML markup. Figure B.3 presents a configuration for generating a set of fuzzed TIFF images. The model does not include extra configuration for automatically running or monitoring test case runs. Information about configuring Peach for these activities can be found at [Eddington 2006].

The DataModel element in Figure B.3 defines the composition and structure of a basic TIFF image. The model is composed of child elements denoting strings, numbers, choices, and references to other models. Each child element has a descriptive and unique name attribute. These child elements also contain a value attribute providing Peach with a default value for the defined field. Some elements also define endianess to direct Peach on data formatting.

An interesting and versatile child element demonstrated in this model is the Blob element. It defines a byte array of arbitrary length. This element is useful when the structure of a section of a file is unknown or the model requires a series of specific data. Child elements presented in Figure B.3 are only a representative sample of data modeling options and not an exhaustive list. A complete list of elements and attributes are available at [Eddington 2006].

Aside from data modeling, Figure B.3 defines a StateModel, Publisher, and a Logger. A StateModel assigns a series of steps the fuzzer must take during the generation of a negative test case. Generically, a state transition diagram for file fuzzing requires writing the negative test case to disk, closing the file, and launching the prescribed application with the test case. Figure B.3 only illustrates writing a negative test case to disk for later testing by invoking the FilePerIteration Publisher.

A Publisher is the mechanism Peach utilizes to generate a negative test case. This mechanism is chosen based upon the input vector being tested. The FilePerIteration Publisher writes every test case to disk with a filename specified by the filename param element's value attribute. Other Publishers are capable of sending traffic over the network or even launching external processes.

The last notable element of the example configuration is the Logger element. It defines the area on disk for storing log files. In this example, all logs are stored in a directory named logs within the directory where the fuzzer was invoked. Log files are uniquely named by appending the fuzzer's start date and time to the base name of the log file. Peach's logging functionality is extensible, however, at this time, the only logging supported is to a directory.

Peach validates the supplied configuration and data model before generating any negative test cases. If an error is found during validation, the fuzzer notifies the tester and ceases operation. After validation, the fuzzer proceeds to select attack heuristics based on the data model's fields. Peach contains many generators and mutators specifically targeting certain data types. One example generator is the UCS Transformation Format 8-bit (UTF8) generator. It is responsible for encoding strings in the UTF8 format and inserting long attack strings encoded as UTF8. Peach continues selecting different fields for attack heuristic insertion and generating different permutations of the data model until all possible combinations are exhausted. This algorithm can be modified by marking fields in the data model as immutable.

Peach is a mature fuzzing framework that has found many software defects. The learning curve can be steep, but the fuzzer can thoroughly tests a program due to its rich data modeling language. The multitude of modeling options combined with different encodings, mutators, and generators covers a wide range of attack heuristics. Peach is definitely one of the best generation fuzzers available today.

The last category of fuzzer, feedback, is a relative newcomer to negative testing. Introduced only in the past five years, feedback fuzzers are slowly evolving. A feedback fuzzer evaluates software from a search space optimization view point with information from static and dynamic binary analysis. Static binary analysis interprets compiled code by extracting program characteristics. Tools, such as IDA Pro or objdump, handle converting a binary representation to intelligible assembly code. Dynamic analysis monitors a running program to profile execution. It is achieved through DBI frameworks like DynamoRIO, PIN, and Valgrind or other debugging interfaces.

Information obtained about code structure and flow from static and dynamic analysis are fed into common search space algorithms such as genetic algorithms or constraint solvers. The fuzzer makes informed decisions about future test cases based upon the feedback from analysis of a currently running test case. The algorithms try to pinpoint problematic values in particular test cases or optimize metrics such as code coverage. These attempts to optimize metrics or solve constraints are an effort to minimize the number of negative test cases generated in order to find a defect.

Vuagnoux [2005] demonstrated one of the first feedback algorithms in 2005 with the fuzzer Autodafé. Its algorithm recedes from primarily focusing on accurate test case modeling combined with a robust set of attack heuristics to an examination of the complexity involved in finding a fault. Fuzzer complexity can be defined by determining the cardinality of the attack heuristics and the number of variables labeled for substitution in a test case model. Since each variable is replaced with each attack heuristic, the complexity of a single fuzzer run is:

$$Complexity = LF \tag{2.1}$$

where L represents the cardinality of the attack heuristics and F denotes the number of variables labeled for substitution. A decrease in complexity then requires either lowering the cardinality of the attack heuristics or eliminating some of the variables labeled for substitution. Removing attack heuristics degrades a fuzzer's ability to elicit a fault, because attack heuristics are predefined based on past observations. These heuristics have proven able to elicit faults from programs in the past, so these values are likely able to elicit faults in other programs [Vuagnoux 2005].

Based on these observations, the Autodafé fuzzer implements an algorithm known as fuzzing by weighting attacks with markers. The concept is to decrease the number of variables labeled for substitution, F, by tracing a program with a debugger before fuzzing. The tracer informs the fuzzer of variables labeled for substitution (markers) used as arguments to unsafe C library function. These functions, for instance strcpy() or strcat(), are notoriously a source of buffer overflows. If a marker is a parameter to an unsafe C library function, then the fuzzer is more likely to elicit a fault by substituting an attack heuristic for that marker. Weights are assigned to each marker based on its frequency of usage as a parameter to an unsafe C library function. All markers are then ranked according to weight and higher weighted markers are substituted with attack heuristics first. Markers assigned a weight of zero are eliminated from the initial fuzzing runs. This classification of markers decreases the complexity by eliminating some markers and prioritizing testing.

Sparks et al. [2007] also reduced negative test case generation to a search space optimization problem by applying genetic algorithms to fuzzing. This application relies on a disassembly of the program under test. The disassembly provides a wealth of information pertaining to possible control flow paths during execution. Their algorithm, Sidewinder, employs a well-known reverse engineering tool for mining a program disassembly, IDA Pro. This application is an immensely useful tool capable of disassembling many different file formats, particularly executables, and synthesizing the binary into intelligible assembly instructions. It also exposes the capability to examine data within a file as different data types such as strings, integers, and even data structures. More importantly, for Sidewinder, IDA is adept at generating control flow graphs from a disassembly. All of this information is exportable to external programs through a Software Development Kit (SDK) [Eagle 2008], which Sidewinder leverages to gather control flow graphs.

Once the control flow graph information is exported from IDA Pro, Sidewinder determines all interesting execution paths. A path is interesting if a node in the control flow path is a vulnerable C function call. An example of a vulnerable C function call is a library function that does not ensure the length of the data being copied is less than or equal to the length of the destination buffer. Functions like strcpy() embody this definition. Also, any function may be labeled as a vulnerable function call by the tester. The function, strcpy(), is only one such example.

Sidewinder prunes subgraphs containing these nodes and paths from external interfaces, such as recv() calls, from the larger control flow graph to guide execution analysis. Up to this point, the algorithm only knows static information about execution paths and possible problematic code sections. To be effective, the fuzzer still requires a method for negative test case generation. It employs a context free grammar as a modeling language to aid in this endeavor.

A context free grammar is a way of formally representing a language through the use of nonterminals, terminals, and productions. Nonterminals can be transformed into a different set of terminals and nonterminals through a rule. A language's terminal symbols are the members of its alphabet which can not be replaced through a production. Transformations, also known as the language's productions, are the rules for substituting a nonterminal or terminal for another sequence of nonterminals or terminals. A string is produced in the language when all nonterminals are replaced with terminals.

Sidewinder encodes test cases as integers corresponding to a series of productions within the defined context free grammar. These encodings are members of a genetic algorithm that evolves the context free grammar, so that the fuzzer can learn to create valid input. This grammatical evolution creates successive populations of integer strings measured by a predefined fitness function. To calculate the fitness of a population member, the fuzzer tracks all node traversals within a program's control flow graph. Based on the recorded traversals, the fuzzer calculates the probability of a test case executing code in one of the next connected nodes on the graph. For each edge in the graph, it calculates these probabilities. The probabilities are only based on node traversals observed during the current generation. After all population members execute, their fitness values are calculated by multiplying all the transition probabilities of the taken transitions for the test case. Sidewinder then tries to increase code coverage by selecting population members with a lower fitness. The tracking of transition probabilities in this manner builds a Markov model of program execution [Sparks et al. 2007].

Sidewinder traces execution by placing breakpoints with a debugger on all entry points in the subgraphs extracted from the IDA Pro disassembly. To decrease execution time, the graphs are broken into two sets. The first set includes nodes leading down paths that could not lead to the execution of an unsafe API call. The second set is the complement of the first set. If execution enters a node in the first set, rejection set, then execution is halted. Otherwise, execution continues until the target vulnerable function call occurs. The breakpoints in the subgraphs serve as informational points for fitness calculations and monitoring points to kill execution. The fuzzer feeds the fitness calculations back into test case generation using the typical genetic algorithm operators to evolve the next generation of cases.

Sidewinder is innovative in its approach to test case generation. It is the first of its kind to rely upon evolutionary techniques to guide negative test case generation. Modeling with a context free grammar yields a sharp contrast from traditional block based modeling approaches. Also, mixing static and dynamic analysis is another novel concept carried throughout feedback fuzzers today. Sidewinder is a historical milestone in fuzzer evolution.

DeMott et al. [2007] enhanced genetic algorithm fuzzing by focusing on evolutionary theory and paralleling white-box evolutionary testing [Takanen et al. 2008]. Evolutionary testing is a code analysis technique based on the principles of genetic algorithm meta-heuristic search [Wegener 2001]. These algorithms analyze a program's source code by breaking it down into a control flow graph. It then references this graph during test runs to calculate the fitness of a particular case [Sthamer et al. 2001]. The makeup of one of these fitness calculations is the summation of a case's approach distance and branch distance. The branch distance equates to the number of branches taken by a case on the control flow graph, while the approach distance equates to the distance from the current value required to execute a particular section of code [Takanen et al. 2008]. In this fitness function, values closer to zero reflect a higher test case fitness.

Jared Demott's fuzzer, Evolutionary Fuzzing System (EFS), borrows the notion of program analysis and evolutionary search, but it shifts the evaluation to the grey-box realm, where the machine code is available but the source code is not. With grey-box testing, the engineer can only examine machine code disassembly to aid in testing. EFS further distinguishes itself from evolutionary testing by focusing on protocol discovery instead of building program specific knowledge into the testing structure. The only specific information pertaining to the program under test is derived from a seed file containing example valid input. This seeding decreases the initial time required to learn protocol basics, so the fuzzer is capable of traversing deeper into the program under test [DeMott et al. 2007].

EFS utilizes the same evolutionary ingredients as a traditional genetic algorithm. The most basic of these ingredients is the notion of a population member. A member, in EFS, is defined as a session with the program under test. Since the fuzzer mostly exercises a program's network interfaces, a session denotes a complete conversation between a client and server. The individual reads and writes composing the session are named legs. These legs are dissected into tokens that contain a type and an actual piece of data. Several token types are available such as text, binary, and length fields. Multiple sessions are lumped into a management structure known as a pool. These structures purely arrange sessions into a logical collection for evaluation purposes. A number of pools containing a number of individual sessions constitutes a generation.

The fuzzer executes a program under test with each session within each pool. It monitors each instantiation for the number of blocks hit in the control flow graph along with program crashes.

After every session in every pool has been executed, each session is evaluated by the number of code blocks exercised to derive the session's fitness. Usual genetic algorithm operators, crossover and mutation, occur with the sessions to derive the next generation.

Also, at configurable generational intervals, the fitness of the logical pools are evaluated. Pool fitness is the summation of the fitness of all sessions comprising the individual pool. When EFS calculates pool fitness, it crosses over and mutates pools to produce the next population of pools. The algorithm for session and pool crossover are the same. Both structures rely on single point crossover for breeding. The addition of elitism into the algorithm affords the most fit session in a generation and the overall best fit pool to be placed directly in the next generation of test cases.

As for mutation, the pool and session algorithms completely differ. For a session, the mutation randomly modifies a token with a predefined attack heuristic. Pool mutation relates to the addition or deletion of a session from the pool. After a generation is computed via crossover and mutation, the fuzzing continues with the same procedures until the system converges or the tester suspends testing.

The EFS fuzzer is a significant step forward in genetic algorithm feedback fuzzer design. It adds more concepts and techniques of genetic algorithms and pulls inspiration from evolutionary testing. The evolution of individual sessions and pools is a novel concept.

Campana [2009] extended feedback fuzzing by applying constraint solving to negative test case generation. Campana [2009] borrows from a technique known as taint tracing, where the flow of data input into a program is monitored throughout execution to determine the different paths taken. A fuzzer named Flayer [Drewry and Ormandy 2007] demonstrated this ability with the Valgrind DBI. Valgrind emulation is extremely useful, especially when tracing data, although, translating a program to an intermediary representation drastically increases execution time.

Fuzzgrind builds upon Flayer's methodology by recording information about the data dependencies leading to different branch points. The tracing information obtained during emulation feeds into a custom Python library responsible for transforming the taint tracing information into a set of constraints. These constraints are then formatted to work with the STP constraint solver library. This library is a decision procedure for bit-vectors and arrays [Ganesh and Dill 2007]. Released in 2007 by Dr. Vijay Ganesh, STP's grammar defines satisfiability equations for which it attempts to solve.

Fuzzgrind leverages the STP library to solve constraint equations causing program execution to steer down differing paths of execution. The algorithm for generating these test cases is based off the work done by David Molnar with SAGE [Godefroid et al. 2008]. This algorithm, combined with the Valgrind DBI, enables the fuzzer to probe deep into the program under test's logic by calculating the values required to exercise different code paths. The algorithm is an innovative technique for

fuzzing. It leverages cutting edge technology to reduce the problem of fuzzing to the problem of constraint solving. This reduction to an NP-complete problem does contain some drawbacks. The most notable being path explosion with sufficiently large programs. This problem is being addressed with STP, and even the SAGE implementation shows promise for large scale program evaluation [Molnar 2009].

This literature review has introduced expected requirements for a fuzzer and a rough fuzzer taxonomy. An understanding of requirements and a method for fuzzer classification is helpful for understanding the evolution of fuzzer design. Fuzzer classifications are not necessarily complex and, as an easy to remember analogy, different fuzzer classifications are roughly comparable to the sport of darts. Mutation fuzzing, dumb fuzzing, constructs a negative test case by inserting random or predefined values into valid test data. This behavior can be likened to a dart thrower trying to hit a bullseye while wearing a blind fold. The player's throws may or may not ever hit the dartboard due to a lack of environmental information. Generation fuzzing, intelligent fuzzing, creates a negative test case by referencing a model of valid test data and inserting random or predefined values into specific portions of the model. The fuzzer selects the portions of code to test by selecting the portion of the model to change. It is akin to a dart thrower being able to see a dartboard for aiming but not adjusting throws to rectify past errors. The thrower is destined to hit the same areas of the dartboard every time. Feedback fuzzing improves upon the other categories by adding a search or learning algorithm. A fuzzer generates test cases based on metrics collected from the execution of previous negative test cases. This resembles a dart player improving by factoring in previous experiences. In the long run, if a player is able to sufficiently learn, then the player should be able to eventually consistently hit the bullseye. In summation, fuzzer classification determines fuzzer composition and complexity. It assists a software tester with fuzzer selection, and the classification can aid in achieving a better understanding of a fuzzer's internals.

Chapter 3

Reverse Engineering

When embarking on intelligently fuzzing an application's file format parsing, the first step is to assess one's access to source code. Often times, access to source code is unavailable, because the individual evaluating the software does not work for the company who developed the product. In these instances, if the application being evaluated is a machine language executable, then the evaluator must reverse engineer the binary to extract information about code structure.

This research targets testing proprietary, closed source Mac OS X applications. Since access to source code, in this situation, is infeasible, the extraction of information about code must occur by analyzing a binary's executable format, determining shared libraries linked against an executable, selecting a shared library to reverse engineer, disassembling the selected library, and finally programmatically extracting any pertinent attributes from the disassembly. To begin this journey, some background in reverse engineering is required. The following briefly reviews reverse engineering and the Mach-O format before walking through an example of gathering information to incorporate into negative test case generation.

Reverse engineering analyzes a physical product or software program to generate a conceptualization of its implementation and component interrelationships [Chikofsky and Cross 1990]. It creates alternative representations of the original, finished product. These representations aid engineers in dissecting a product for comprehension of its nuances and intricacies. Alternative representations, for the purpose of this research, supply fuzzing algorithms with supplemental information about a program under test.

The practice of reverse engineering originates from hardware engineers disassembling physical products. Hardware disassembly catalogues a product's internal layout, circuits, and interconnections. Government agencies and industry competitors quickly adopted these practices to find design flaws or catalogue competitor features, functionality, and cost breakdown. While the origins of

reverse engineering are firmly rooted in hardware, software reverse engineering has gained popularity over the past twenty years [Chikofsky and Cross 1990].

Reverse engineering applied to software embodies many of the same objectives of hardware reverse engineering. However, instead of examining physical components, software reverse engineering analyzes binary data for comprehension of control flow, data flow, and data structure. This analysis is not restricted to software programs, and it may be applied to analyzing arbitrary binary data such as proprietary file formats or network protocols. Arbitrary binary analysis of this data only precisely formulates a model of the data's structure. For reversing software programs, an alternative view of data flow, control flow, and program structure is necessary to foster understanding of machine code.

The accuracy of reverse engineering software programs is contingent on the disassembly algorithm utilized. Two well-known methods exist for translating an executable or shared libraries' machine code into assembly language. The first, Linear Sweep algorithm, begins by deciphering an executable's file format and extracting the text section start and end addresses from the file's metadata. The algorithm then reads the bytes stored at the starting address and decodes those bytes into an assembly language instruction. An executable's architecture (x86, MIPS, PowerPC, etc.) determines an assembly instruction's encoding and the number of bytes comprising the encoding. For instance, Microprocessor without Interlocked Pipeline Stages (MIPS32) assembly instructions are 32-bit fixed-width instructions [Sweetman 2006]. This means the Linear Sweep algorithm should always read four bytes of data from the text section to decode an instruction.

On the other extreme, x86 instructions are variable length alignment independent. Instruction length variation complicates the disassembly process, because the disassembly algorithm must calculate instruction length. For x86 instructions, the first few bytes of an encoded instruction represent a prefix and an opcode. These encoded values together determine an instruction's encoded size. Once the disassembly algorithm decodes these values, the instruction's total encoded length can be determined by consulting a look up table keyed on prefix and opcode. This description simplifies decoding x86 instructions only to illustrate the added complexity of supporting variable length instruction sets [Intel 1999].

Once the encoded bytes are translated into an assembly instruction, the Linear Sweep algorithm reads and decodes the next sequential bytes. It continues linearly sweeping through an executable's text section decoding instructions until it encounters the text section's end address. Linear Sweep efficiently translates machine code into intelligible assembly instructions, but it often misinterprets data interspersed throughout the text section. Compilers commonly optimize code in many ways, such as generating jump tables, and these optimizations cause data to be interspersed in the text section. Since the algorithm linearly sweeps the entire text section without interpreting assembly instruction semantics, it cannot differentiate between text and data. The objdump utility, found on

most Linux operating systems and in the GNU's Not Unix! (GNU) binutils package, implements the Linear Sweep algorithm [Linn et al. 2003].

The Recursive Traversal algorithm combats this weakness of the Linear Sweep algorithm by interpreting semantics of assembly language instructions altering control flow. It starts, like the previous algorithm, by determining an executable's file format and extracting text section start and end addresses. The algorithm then reads the sequence of bytes at the start address and decodes them into an assembly instruction. It continues linearly reading and translating bytes until the algorithm detects a branch instruction. When encountered, instead of reading the next sequential instruction, the algorithm attempts to follow all possible control flow paths from the branch instruction (target and fall-through) and continues disassembling at each new path. It ceases operation once all control flow paths for all branches have been decoded [Schwarz et al. 2002].

Recursive Traversal, while an improvement upon the Linear Sweep algorithm, is also fallible. An assembly instruction sequence can disrupt the algorithm by directly altering a function's return address before issuing a return from a function call. If a disassembler does not account for this nuance and relies purely on calling conventions (e.g. cdel, stdcall, or fastcall), then an incorrect disassembly can result [Eagle 2008]. Recursive Traversal also has trouble following indirect jumps if other heuristics are not applied to detect these constructs. Nuances such as these happen less frequently than compiler optimizations effecting the Linear Sweep algorithm which make disassemblers implementing the Recursive Traversal algorithm preferable.

IDA Pro, an industry standard disassembler and debugger, implements a variation of the Recursive Traversal algorithm. Many engineers regard it as the best in class disassembler for binary and malware analysis. It supports multiple executable formats for multiple architectures. The information generated by IDA Pro also easily exports to other programs through two scripting interfaces. The first, IDC, resembles the C programming language, and the second, IDAPython, extends the Python scripting language with Simplified Wrapper Interface Generator (SWIG) bindings [Eagle 2008]. Both extensions support automating common tasks inside IDA Pro along with retrieving attributes from a program disassembly. This disassembler overshadows many other disassemblers and disassembly libraries like udis86, distorm64, and Metasm. Its superior disassembly algorithm, vast architecture support, and multiple scripting interfaces make it the best choice for inclusion in this dissertation's fuzzing framework. Other disassemblers and libraries provide less flexibility and lack support for many architectures and platforms.

3.1 Mach-O Format Analysis

Fuzzing frameworks have historically targeted programs running on popular operating systems like Windows, Unix, and Linux. The first fuzzer, from Dr. Barton Miller's research group at the University of Wisconsin, Madison, tested Unix utilities [Miller et al. 1990]. Later fuzzers, such as notSPIKEfile and Peach, started by supporting Linux and Windows. Even new fuzzers like EFS started by testing Windows applications. Due to the increasing popularity of the Mac OS X operating system and the lack of compatible fuzzers, this research develops a fuzzing framework with an emphasis on testing Mac OS X applications. Since Mac OS X applications are the primary focus of this research, its executable file format must be explained to exhibit how to extract pertinent information from Mac OS X binaries.

An operating system designer defines an executable file format with a formal specification. These specifications describe required composition and structure of compiled executables or shared libraries. An operating system then includes functionality, known as a loader, to recognize the defined format, parse the data, layout the program in memory, and begin execution of the machine code. Many different standards exist defining executable formats. Microsoft supports the Portable Executable (PE) format for its programs and dynamically linked libraries on the Windows operating system. The Executable and Linkable Format (ELF) and a.out formats are commonplace on Linux and Unix. As for Mac OS X, most executables conform to the Mach-O format.

A Mach-O file is an object file storing machine code, data, and metadata describing the object file's contents. Machine code results from compiling a language, such as C, C++, or Objective C, and linking it against shared libraries. During compilation, the code looses most human readable constructs and symbols especially when symbol tables are stripped from the binary. A disassembler attempts to reconstruct contextual information and semantics by translating machine code into assembly language. Individuals auditing programs released from companies with whom they have no association must rely on information from a disassembler to help understand control flow and code structure.

Mach-O file structure consists of three major regions, the first of which begins with a header. The header region starts with a magic number, 0xCAFEBABE, to differentiate it from other file formats. The Mac OS X loader detects this magic number when loading an executable and recognizes the format as a Universal Mach-O file. As a side note, Java class files are also identifiable by the 0xCAFEBABE magic number [Apple 2009].

The Universal Mach-O is one of two variations of Mach-O files. The other format is known as a Standard Mach-O. The universal format is the predominant format and a consequence of backward compatibility support for standard Mach-O files of different architectures. Earlier versions of Mac

OS X only supported the PowerPC architecture. Later versions ported the operating system to the ubiquitous x86 architecture. In order to guarantee operation of newer applications on older systems, Apple created the universal format. It concatenates x86 and PowerPC Mach-O files into a single object file.

The next region of a Mach-O file defines a set of load commands. These commands describe object attributes like the location of symbol tables, dynamic linker for loading the object, and addresses of shared library initialization routines. In general, object file load commands can be extracted by the otool utility packaged with Mac OS X. To list all commands, otool must be executed with the -l option.

More importantly for this research, the load commands contain the names of libraries linked against the object file [Iozzo 2009]. The fuzzing framework resulting from this research selectively monitors execution of shared libraries linked against an object file. While the load commands specify these libraries, output from otool's -l option does not succinctly display all shared libraries. Fortunately, otool supports another option, -L, which only lists shared library load commands. This listing completely displays libraries for the shared object itself, but each shared library may be linked against other shared libraries. In order to get a full listing, a recursive query must be performed against all shared libraries linked against the original object file.

Since determining and selecting a shared library to evaluate is an important component for the fuzzing framework defined in this research, the framework contains a tool, '`thor tools:otool`', in support of this task. This tool wraps the otool command and recursively interrogates all shared libraries linked against an executable. The recursive interrogation runs otool -L against each shared library and adds newly discovered libraries to the list of libraries yet to be interrogated. Once all libraries are scrutinized, the tool prints a full listing, and then the software tester can choose a library for testing. The inclusion of this tool within the framework decreases the amount of time required to select a shared library for monitoring. It also presents a unified fuzzing environment where tasks associated with fuzzing can be carried out along with the actual fuzz testing. For more information on this tool, its usage, or any tool included with the fuzzing framework, please consult Appendix C.

The last region of a Mach-O file is the data region. It is broken into multiple segments encapsulating multiple sections. Segments define different pieces of an executable such as text and data. A section within a segment is logically specific data pertaining to an overall segment. For example, the text segment contains a _text section holding an application's machine code. Another example is the data section encapsulating a _data segment holding initialized mutable variables [Apple 2009].

3.2 Mach-O Disassembly

Once a shared library is chosen from the list of libraries linked against a Mach-O executable, disassembly of the shared library must occur to begin the attribute extraction process. As an example of this process, Figure 3.1 introduces a fictitious, errant Objective-C library to parse a file in a Type-length-value (TLV) format. This format prevails in many network protocols such as Simple Network Management Protocol (SNMP) and Intermediate System-to-Intermediate System (IS-IS). To compile this library as a shared library on Mac OS X, the -dynamiclib flag must be supplied to GNU Compiler Collection (GCC) during linking.

```
#include <Foundation/foundation.h>
#include <Foundation/NSFileHandle.h>

void vulnerable_parser(NSFileHandle *fileHandle, UInt32 filesize)
{
    char value[24];
    int type;
    int length;
    while([fileHandle offsetInFile] < filesize-8)
    {
        [[fileHandle readDataOfLength:4] getBytes:&type];
        [[fileHandle readDataOfLength:4] getBytes:&length];
        switch (type) {
            case 16:
                [[fileHandle readDataOfLength:length] getBytes:&value];
                break;
            default:
                if(length < 24)
                    [[fileHandle readDataOfLength:length] getBytes:&value];
                break;
        }
    }
}
```

Figure 3.1: C Code TLV Parser Example

As background for the TLV format, three different fields constitute a TLV encoding. The first, type field, characterizes data stored in the value field. It specifies what type of data is stored in the value field. The second, length field, dictates the total size of data stored in the value field, and the last field stores the actual data. TLV encoding exists to store variable length data for transmission, storage, and parsing.

The example library exhibits a common error found in many TLV parsing libraries. The function, vulnerable_parser(), consumes TLV encodings one at a time from an already opened file. For each encoding, it first reads the type and length fields to determine the case to execute in the switch statement. If the type field equals sixteen, then the code blindly trusts the length read from the file resulting in possibly over flowing the value buffer on the stack. Blindly trusting user supplied lengths is a common mistake of many parsing libraries.

Another more subtle error also resides in the example library. If the type field does not equal sixteen, then the code validates the length field to ensure it is less than the value buffer's size. If the length is not less than twenty four, then the parser abstains from copying any data from the file into the value buffer. It then continues reading type and length information. Continuing parsing when the proceeding length value is not less than twenty four is errant, because the parser consumes type and length information from an intended data field. The parser is now misaligned for the remainder of file parsing.

These errors are especially easy to identify when source code is available and the code lacks complexity. However, often times, source code can not be consulted. In these cases, the binary must be evaluated by first translating the machine code into readable assembly language. Figure 3.2 presents snippets of disassembly from the compiled library presented in Figure 3.1. The assembly language shown in the figure is taken from the IDA Pro disassembler.

```
Location         Disassembled Instruction
<snip>
__text:00000E40 _vulnerable_parser proc near
__text:00000E40
__text:00000E40 var_4C = dword ptr -4Ch
__text:00000E40 var_48 = dword ptr -48h
__text:00000E40 var_44 = dword ptr -44h
__text:00000E40 var_40 = dword ptr -40h
__text:00000E40 var_3C = dword ptr -3Ch
__text:00000E40 var_38 = byte ptr -38h
__text:00000E40 var_20 = dword ptr -20h
__text:00000E40 var_1C = dword ptr -1Ch
__text:00000E40 arg_0  = dword ptr  8
__text:00000E40 arg_4  = dword ptr  0Ch
__text:00000E40
__text:00000E40 push    ebp
__text:00000E41 mov     ebp, esp ; Function Prologue
__text:00000E43 push    edi
__text:00000E44 push    esi
__text:00000E45 push    ebx
__text:00000E46 sub     esp, 92
__text:00000E49 call    $+5     ; PIC
__text:00000E4E pop     ebx
<snip>
__text:00000E80 mov     eax, ds:(msg_a0ffsetinfile - 0E4Eh)[ebx] ; message
__text:00000E86 mov     [esp+4], eax
__text:00000E8A mov     [esp], esi
__text:00000E8D call    _objc_msgSend
<snip>
```

Figure 3.2: Assembly Code TLV Parser Example

IDA Pro displays assembly instructions in Intel syntax instead of AT&T syntax. Intel syntax is discernible from AT&T syntax by the absence of percent signs prefixing register names and the usage of square brackets for addressing. Determining assembly language syntax is important, because AT&T syntax displays source operands on the left followed by destination operands. Intel syntax

formats assembly instructions opposite of AT&T syntax. Incorrectly interpreting assembly formats completely changes assembly instructions semantics [Eagle 2008].

As mentioned earlier, IDA Pro implements a variation of the Recursive Traversal disassembly algorithm. While this algorithm outperforms the Linear Sweep algorithm, IDA Pro's implementation suffers from a few peculiarities when disassembling Objective-C Mach-O binaries. IDA Pro, versions prior to the recently released 6.2, does not completely disassemble Objective-C class method calls, EIP-relative data addressing, and identifying some functions. However, IDA Pro is an interactive disassembler and a tester can fix these issues by automating or manually altering the generated disassembly. Miller and Zovi [2009] discuss techniques and present scripts to resolve these failures.

Most Mac OS X libraries and executables are programmed using Objective-C. This language is an object oriented programming language relying on message passing to invoke object class methods. Message passing complicates disassembly, because class methods are not directly invoked. Instead, messages are sent to objects with a function from the objc_msgSend() family of functions. The message sent reflects the class method to invoke. Generally, calls to objc_msgSend() are responsible for invoking class methods, and the parameters to this function are the object to receive the message, message to send to the object, and any parameters required by the object's message handling function. IDA Pro fails to follow Objective-C message passing because intermediate calls to objc_msgSend() invoke class methods and the parameters to objc_msgSend() must be examined to determine which class method is invoked [Miller and Zovi 2009].

Figure 3.2 illustrates an example of this message passage convention. At address 0x0E8D in the figure a call is made to objc_msgSend() with the offsetInFile message. This function call invokes the NSFileHandle offsetInFile method to retrieve the current offset of the file handle. Comparing the disassembly to the source code in Figure 3.1, this portion of disassembly maps to the while loop condition.

EIP-relative data addressing is an adaptation of RIP-addressing defined by the x86_64 Application Binary Interface (ABI). These addressing modes access data by offsetting from the current instruction pointer [Matz et al. 2010b]. Addressing in this manner assists in generating position independent code, because data accesses are no longer reliant on fixed memory addresses. For IDA Pro, relative addressing from the instruction pointer causes the disassembly algorithm to miss jump tables referenced by this addressing. Since the jump tables are not found, portions of code fail to disassembly. These jump tables can be corrected by manual inspection and IDA Pro automation [Miller and Zovi 2009].

Missed functions are another unfortunate occurrence when disassembling binaries. These functions may elude IDA Pro's disassembly algorithm due to a variety of reasons. One such instance may be that unrecognized jump tables indirectly access sections of code. If IDA Pro fails

to recognize a jump table branch, then the disassembly algorithm will not descend into the code section to disassemble it. Fortunately, additional functions can be found after IDA Pro disassembles a binary by searching for assembly instructions indicating a function prologue. Instructions saving the stack frame base pointer, ebp, and immediately resetting its value are hallmarks of function prologues [Hotchkies 2008].

The effects of Address Space Layout Randomization (ASLR) also must be addressed while reverse engineering Mach-O binaries. ASLR randomizes the address space layout of a program when it is loaded into memory. This randomization prevents malicious code from returning to known memory locations to leverage existing code or begin executing injected code [Shacham et al. 2004]. On Mac OS X, software upgrades or executing the update_dyld_shared_cache program randomizes dynamic libraries. Files in the /var/db/dyld/ directory detail the randomized offsets. While these libraries remain at fixed offsets until updates occur, the offsets are different from machine to machine.

Combating randomization is unnecessary if the goal of reverse engineering is only to understand a library. However, if monitoring code execution on many different machines is desirable, then the random offsets must be eliminated to guarantee an instruction executed at a particular address is the same across multiple machines. With the disassembler, the best way to minimize the effects of ASLR during attribute extraction is to set the loading segment and offset in IDA Pro to be zero. Setting these values to zero offsets all disassembled instructions from address zero. This offsetting now provides a measurable displacement from any base address. Later, Chapter 4 clarifies the importance of using addresses of basic blocks and functions as offsets from zero.

Appropriately addressing IDA Pro deficiencies before attribute extraction improves attribute accuracy. Incorrect disassembly misrepresents the executing code. This misrepresentation hinders proper measurement of code execution and calculation of fitness measures. Errors at this stage of fuzzing ripple throughout all proceeding stages.

3.3 Attribute Extraction

IDA Pro internally represents disassembled programs with several logical structures. At the top of these logical structures is one referred to as a segment. Segments are akin to a segment and section as described during the discussion of the Mach-O file format. This structure contains similar portions of an object file. Examples of segments are text, data, and Block Started by Symbol (BSS). A disassembler, IDA Pro, determines the start and end of a segment by parsing an object file's header. A person performing analysis of an object file may create segments manually or programmatically as well.

Focusing on the text segment of an IDA Pro disassembly, the next and most important logical structure is a function. Functions are a collection of assembly instructions grouped to perform an operation. Functions contain a prologue and an epilogue which create and destroy a stack frame for local variable storage. Parameters to functions may also be stored on the stack prior to calling a function or passed to the function by storing the parameter in a predefined register. Argument passing and stack frame allocation methods depend upon the architectures ABI which defines function calling conventions. Different programming languages may also implement different calling conventions. An example of different calling conventions is illustrated by comparing how the x86 and x86_64 ABI's define passing arguments to functions. For the x86 cdel calling convention, function arguments are passed on the stack by pushing them onto the stack. The order they are pushed is from right to left meaning the first argument to a function is on top of the stack. For x86_64, depending upon the argument type, most arguments are first stored in a register (e.g. rdi, rsi, rcx, r8, r9), and any leftover parameters after register allocation are stored on the stack [Matz et al. 2010b]. Due to these possible differences, when analyzing object files and extracting attributes, architecture ABI's and calling conventions must be consulted to ensure the accuracy of extracted information and extraction algorithms must be altered to account for these differences.

A number of properties are automatically associated with a function by IDA Pro. Two attributes are the function's start and end addresses. These addresses define the location of the function's first instruction and the address of the instruction immediately following the function. IDA Pro also calculates the stack size for a function's local variables and other qualitative attributes like labeling a function as a library function. Library functions are functions identified by IDA Pro based on its Fast Library Identification and Recognition Technology (FLIRT) signatures. These signatures identify code sequences by pattern matching. For example, a FLIRT signature may match code with functionality similar to strcpy() [Eagle 2008].

IDA Pro also logically dissects a function into one or more chunks. A compiler creates a chunk to optimize the location of frequently executed code. Less frequently executed code is moved to memory regions less likely to be swapped into memory while the program is running. Code reorganization guards against frequently executed code being continuously swapped in and out of memory [Eagle 2008]. Only certain compilers optimize code into chunks, and, in most instances, a single chunk constitutes a function. Chunk attributes include a start address, end address, and a list of basic blocks.

A basic block is a sequence of assembly instructions with exactly one entry point and one exit point. Execution of the first instruction in a basic block guarantees execution of all subsequent instructions, because the only exit point occurs at the basic block's end. A basic block's exit point transfers control to another basic block. These blocks, like functions, have start and end address

attributes automatically associated with them. The calculation of a basic block's size, in terms of the number of assembly instructions composing the block, is also capable of being computed by a scripting interface.

All of these logical structures and associated attributes are exportable from IDA Pro through either the IDC or IDAPython scripting interface. Both interfaces are well documented, yet the IDAPython interface's extension of the Python scripting language makes it more flexible. Due to this flexibility, many scripts and even a framework have been developed in Python to aid in reverse engineering and debugging binaries with IDAPython. The most popular of which is the PaiMei library written by Pedram Amini. This library contains a module, pida, that iterates over functions, chunks, and basic blocks of a disassembly's text segment. While iterating over these structures, it extracts the automatically associated attributes and calculates additional pertinent attributes. These calculated attributes are information such as the number of arguments to a function, total size of those arguments, number of local variables in a function, size of those local variables, and the total size of a function's stack frame. The pida module then stores this information in a file by pickling the Python objects containing those attributes [Amini 2006].

Pickling or marshaling converts a Python object to a binary format suitable for storage. The binary format easily and seamlessly stores the objects maintaining their structure for later reuse. However, the format defined by Python's pickling routine is incompatible with other programming languages. It is not possible to pickle a Python object, easily read the pickled format, and decode it into an object represented in another language. As such, leveraging the PaiMei framework to extract disassembly attributes potentially locks development into using Python. This Python dependency is undesirable, because other tools written in other languages may benefit from information extracted from IDA Pro. By formatting extracted information in a language agnostic markup, the information can then be shared between individuals without access to IDA Pro, and, other tools can be developed in a variety of languages.

To tackle this issue, alternatives for Python object serialization were researched. One format satisfied the requirement of a language independent representation with broad adoption. This language, YAML Ain't Markup Language (YAML), is a"human friendly data serialization standard" supported by C, C++, Ruby, Python, Perl, and other languages. YAML is human friendly, because it is human readable and expressive. Formatting disassembly information in YAML facilitates information sharing and development of tools in other languages [Ben-Kiki et al. 2009].

This dissertation altered the pida module of PaiMei to serialize attribute information in YAML. This alteration produced a fork of the original code and a companion Ruby language library capable of parsing serialized pida Python objects into Ruby objects. Figure 3.3 illustrates a snippet of attribute information extracted from the example TLV parsing library and serialized in YAML.

```
!fuzz.io/Object
disName: /Users/rseagle/Desktop/dissertation/sharedLibrary/tlvparser.dylib
functionList:
- !fuzz.io/Function
  argSize: 8
  chunkList:
  - !fuzz.io/Chunk
    blockList:
    - !fuzz.io/Block
      branchFrom: []
      branchTo:
      - '0xf26'
      - '0xf55'
      endEA: '0xe80'
      numInstructions: 22
<snip>
  frameSize: 112
  isImport: false
  localVarSize: 104
  name: _vulnerable_parser
  numArgs: 2
  numChunks: 0
  numLocalVars: 8
  startAddress: '0xe40'
<snip>
```

Figure 3.3: YAML Export TLV Parser Example

As shown, the serialized information begins by defining a domain, fuzz.io, and a top-level tag, Object. The domain definition prevents collisions with other YAML specifications, and the Object tag encapsulates all other tags. The next line specifies the fully quantified file name of the disassembled library. As will be seen in Chapter 4, the original full path name identifies the library as it is loaded into memory during emulation. This identification helps determine the address of the shared library in memory and neutralizes the effects of ASLR.

The next entries in the file encode the lists of functions, chunks, and blocks found during disassembly. These lists correspond to the logical IDA Pro structures discussed previously. Each YAML Function, Chunk, and Block contain attributes profiling their assembly counterparts. For instance, the depicted function, vulnerable_parser(), has a total frame size of 112 bytes and 104 bytes of local variable storage. Function attributes, such as these, will be included in fitness calculations in later chapters.

The task of disassembling and exporting attribute information of an object in YAML is as important as determining the shared libraries linked against an executable. Since this task is important for fuzzer configuration, the fuzzing framework, described in the next chapter, includes a tool, '`thor tool:disassemble`', for disassembling and extracting attributes. This tool wraps IDA Pro's command line interface and runs a modified version of the pida IDAPython script. This modified version is suitable for extracting information from an x86 binary and all information except

attributes about function arguments for x86_64 binaries. For usage information, please consult Appendix C.

This chapter introduced concepts pertaining to the evaluation of closed source software. It reviewed common disassembler algorithms and discussed the preferred Mac OS X shared library format. A fictitious TLV parsing shared library demonstrated the steps required to extract attributes from machine code with IDA Pro and IDAPython. This demonstration explained extensions made to the popular reverse engineering framework PaiMei to support language agnostic information sharing with YAML. Future chapters rely upon processing the attributes extracted from IDA Pro to direct fuzzing algorithms.

Chapter 4

Mamba Fuzzing Framework

Binary reconnaissance, as described in Chapter 3, is the first step in fuzzing applications with a genetic algorithm. The next step is to leverage data collected during reverse engineering, correlate it with dynamic analysis, and have a genetic algorithm measure performance based on the correlation. To facilitate measuring a negative test case's performance based on static and dynamic analysis, a fuzzing framework must first fulfill most of the requirements derived in Chapter 2, and additionally support processing information exported during binary analysis along with tracing program execution. An analysis of preexisting fuzzing frameworks determined that the best option for fulfilling these requirements is to develop a new framework. This new framework, Mamba, concentrates on testing the robustness of a Mac OS X application's file parsing routines. It implements general feedback fuzzer functionality and includes multiple feedback fuzzing algorithms.

Mamba is an object oriented framework designed to create a rich environment for application testing. It is written in the Ruby scripting language which promotes rapid development and eventual cross platform support. Yukihiro Matsumoto developed Ruby around 1995 and adoption of the language has significantly increased since its introduction. Ruby is an interpreted language, and its interpreter and core are written in C. All major operating systems, Mac OS X, Windows, and Linux, support this language, and several alternative implementations of the original standard have been developed and adopted. By implementing a framework in a scripting language like Ruby, many low-level implementation errors are nullified and implementation time is decreased. For instance, Ruby mollifies most memory management problems, because it handles freeing memory with its own garbage collection algorithm.

Mamba's architecture conforms to traditional object oriented design principles. The framework implements a base fuzzer class which all other fuzzers inherit and extend. Other common fuzzer requirements, random attack heuristics, execution monitoring, logging, and reporting, are handled

by their own classes. The base fuzzer class then includes instances of the other classes to gain their functionality. This architecture resembles the architecture of Sulley, a popular Python fuzzing framework.

At first blush, extending Sulley may have proved beneficial. However, at the onset of this research, Sulley lacked adequate support for file fuzzing. Sulley's support of Mac OS X also languished behind its Windows support. The creation of a new framework provided tangible research in fuzzer design while improving file fuzzing on Mac OS X.

The version of Mamba released by this dissertation, version 1.0.0, delivers five fuzzers. These fuzzers are Mangle, Simple Genetic Algorithm (SGA), Byte Genetic Algorithm (BGA), Simple Genetic Algorithm Mangle Mutator (SGA-MM), and Cross-Generational Elitist Selection, Heterogeneous Recombination, and Cataclysmic Mutation Genetic Algorithm (CHC-GA). Each fuzzer is capable of being executed in one of two operational modes, centralized or distributed. This chapter only discusses Mamba's installation, configuration, centralized operation, and architecture with the Mangle fuzzer. Chapter 5 describes the included genetic algorithm fuzzers, and Chapter 6 depicts Mamba's distributed model.

The source code for Mamba is available at `https://github.com/rogwfu/mamba-refactor`. It can be installed by cloning the source directory with the git version control tool and then installing the package via Ruby's package management system, RubyGems. This means the user must change directory into the cloned repository and run the '`rake install`' command. During installation, Mamba automatically installs all required Ruby dependencies, downloads mongodb and rabbitmq, and compiles a special version of the Valgrind emulator for execution monitoring as describe in Section 4.4*. Common development tools, gcc, svn, curl, tar, and autotools, are assumed to be installed on the target Mac OS X system. The installation also assumes Erlang is installed (preferably version R14B). Mamba also installs a custom Ruby utility, opener.rb, to support test case delivery.

Once installed, a fuzzing environment is created through the mamba command line utility. It has many options configurable with command line arguments specified by flags. These include setting the application to fuzz (-a), fuzzer type (-y), and test case time out (-t) among other parameters. Running '`mamba help create`' displays a full listing of available options. Additionally, Figure C.4, in the command reference appendix, shows the currently supported options.

For now, the fastest route to starting a fuzzing environment is by running '`mamba create <fuzzer-dir>`' where `<fuzzer-dir>` is a directory to create for a fuzzing environment. Once run, the framework creates `<fuzzer-dir>` and populates it with seven directories (configs, databases, disassemblies, logs, queues, tasks, tests). Each directory maps to a fuzzer functionality. The configs

*A dependency exists on another Ruby package developed during this research. It, Plympton, is available at `https://github.com/rogwfu/plympton-refactor` and is installed in the same manner as Mamba. This package must be installed prior to installing Mamba.

directory holds global and fuzzer specific YAML configuration files. The databases and queues directories contain a database and messaging queues for distributed fuzzing as described in Chapter 6. YAML files generated by IDA Pro attribute extraction defined in Chapter 3 should be stored in the disassemblies directory. Logs of fuzzer runs are found in the logs directory, and generated negative test cases are stored in the tests directory.

The tasks directory is a special directory. Mamba pre-populates this directory with three files containing Ruby code. Each file, distrib.thor, fuzz.thor, and tools.thor, defines multiple functions for interacting with the fuzzing environment via the command line. Functions exist to start and stop a fuzzing job or start and stop the components of the distributed environment. As mentioned in Chapter 3, tools also exist to help with shared library selection and disassembly. Mamba includes these functions to provide the tester with a complete fuzzing environment.

Each function is written with the help of the Ruby package Thor. Thor implements an interface for scriptable tasks similar to GNU build system's make command. To retrieve a list of valid tasks for the environment, run '`thor -T`' in the directory above the seven newly created directories. To execute any given task, run '`thor <task-name> <options>`'. For more information about a task or to list a task's options, run the command '`thor help <task-name>`'. As shown in Figure C.1, tasks are grouped in namespaces based on similar functionality.

The most important tasks to start and stop fuzzing are fuzz:start and fuzz:stop. When started, Mamba creates a file, MambaFuzzingFramework, in the top-level directory. The process id of the currently running fuzzer is stored within this file. The fuzz:stop thor task parses this file to determine the correct Ruby process to kill. Now that a basic overview of installation and usage has been conveyed, the remaining sections of this chapter depict configuration of an instance of the Mangle fuzzer along with walking through Mamba's architectural design.

4.1 Configuration

Once a fuzzing environment is created with the mamba command line utility, the next step in the fuzzing process, prior to starting the fuzzer, is to configure it. During the initial creation of a fuzzing environment, Mamba creates a configs directory where two configuration files in YAML format reside. The first file, fuzzer.yml, configures global fuzzer parameters. This file is present for all fuzzers regardless of type. Figure 4.1 illustrates an example of a global configuration file.

YAML format stores these configurable parameters in key-value pairs. Fields are separated by a colon (:) with the key on the left-hand side of the colon and the value on the right-hand side. Mamba parses this file during startup and stores the key-value pairs in a hash. The contents of this

hash are initially used by Mamba to setup an instance of a fuzzer. The six parameters shown in Figure 4.1 are all standard parameters applicable to all fuzzers.

```
---
:app: /Applications/Preview.app/Contents/MacOS/Preview
:executor: appscript
:os: darwin10.7.0
:timeout: 5
:type: Mangle
:uuid: fz70f8eba2e26611e0b2720026b0fffee7
```

Figure 4.1: Mamba Global Configuration File

The first parameter, app, configures the application to test. In this example, the fuzzer is testing the Preview application which is a PDF viewer. The value of this parameter must include the full path to the executable. The next parameter, executor, defines the delivery mechanism for negative test cases. A delivery mechanism determines how a test case is sent to a program under test for processing. Currently, two different mechanisms are supported, and these are Applescript and the Command-line Interface (CLI). Section 4.3 provides additional details on executors and test case delivery mechanisms.

Directly following the executor parameter is the os parameter. This parameter is strictly for informational purposes. The example shows a value of darwin10.0.7 which corresponds to the Mac OS X 10.6 kernel version. After the os parameter, the timeout parameter determines the amount of time in seconds an executor should wait until an individual negative test case run should be terminated. A timeout value of five, as shown in Figure 4.1, runs a test case for five seconds before terminating the application.

The next parameter, type, assigns a particular fuzzer type to use during testing. This configuration file sets the fuzzer to be Mangle. If a fuzzer type is not specified when using the mamba command line utility during initial creation, then Mamba defaults to the centralized Mangle fuzzer. When the fuzzer is started with the fuzz:start thor task, Mamba parses the type field and checks if the currently installed framework supports this type. To check the type, Mamba queries the framework with the const_defined() Ruby Module method. If the query successfully returns, then the framework allocates and initializes a new instance of the fuzzer type by acquiring a reference to its class with the Ruby Module function const_get(). The newly allocated type specific fuzzer then calls its fuzz method to begin testing. Metaprogramming, in this manner, defines a design pattern for development of additional fuzzers. When a new fuzzer is added to the framework, the only requirements are for it to inherit from the base fuzzer class, define and parse YAML type specific configuration, and define a fuzz method for the fuzz:start task to call.

The last parameter in the fuzzer.yml file is the Universally Unique Identifier (UUID). For each fuzzer created, a UUID is generated by querying the operating system for the current time. This time is then used to seed a secure random number generator. The value returned by the secure random number generator is then appended with the characters 'fz' to yield the final UUID. While it is possible for two fuzzer instances to be created at the same time on different machines, the UUID is only meant to be locally significant when running a fuzzer in distributed mode. Chapter 6 provides more details on the usage of the UUID.

The second file within the configs directory designates fuzzer specific parameters. The name of this file and the key-value pairs within it change depending on the fuzzer type configured in the global configuration file. The file's name equals the fuzzer type name appended with '.yml'. Figure 4.2 displays an example configuration file, Mangle.yml, for the centralized Mangle fuzzer.

```
---
Number of Testcases: 1024
Header Size: 1024
Basefile: tests/test2
Default Offset: 0
```

Figure 4.2: Mangle Fuzzer Configuration File

The first parameter in this file sets the number of test cases to generate and run. In this case, 1024 files will be tested with the Preview application. The next parameter, Header Size, configures Mangle to read and replace 1024 bytes of the base test case file for each generated negative test case. This base test case file is configured to be found in the tests directory with the filename test2 by the Basefile parameter. The last parameter, Default Offset, extends the original Mangle algorithm, as presented in Chapter 2, by allowing Mangle to skip to a file offset before reading and altering the number of bytes dictated by the other parameters. In Figure 4.2, the offset is set to the start of the file (0), but the offset value can be any offset in the base file as long as the offset plus the header size is less than the total base file's size.

Classes implementing new fuzzers handle generation and parsing of these fuzzer specific configuration files. Offloading generation and parsing of these files to their own classes decouples base fuzzer functionality from specific fuzzer functionality. An algorithm can now be independently developed and configured as seen appropriate by the algorithm's author without interfering with global configuration parameters. This flexibility is possible by mandating each fuzzer define their own generate_config() class method for the creation of a configuration file. These type specific files are then read by the fuzzer's type specific class during initialization.

Now that the basics of configuration have been stated, the remainder of this chapter articulates the functionality implemented by Mamba. As a reference, Figure 2.1 in Chapter 2 depicts a suggested fuzzer classification scheme. Mamba's software architecture adheres to this classification scheme by creating classes corresponding to each required component. These classes provide functionality for attack heuristics, an executor, a monitor, a logger, and a reporter. Depending upon the fuzzing algorithm implemented in the framework, additional components, such as a search algorithm, are also included.

4.2 Attack Heuristics

Attack heuristics are the set of malicious inputs to substitute into a test case before executing the program under test with a negative test case. These heuristics can be simple such as adding a variable number of random bytes or more precise like inserting known troublesome values. In either case, a model of a test case is not required, but the latter method benefits from data modeling. Since this version of Mamba does not support data modeling, the attack heuristics are simplified to randomly generated data.

These random values are created by the RFuzz RandomGenerator library by Zed A. Shaw [Shaw 2006]. This library is capable of producing arbitrary length, random sequences of bytes, floats, ints, and base64 encoded data. It implements the ArcFour algorithm documented by Bruce Schneier [Schneier 1995] and also defined in a Request For Comments (RFC) draft [Kaukonen and Thayer 1999] for random generation. Once the ArcFour algorithm creates the random data, the RandomGenerator Ruby class, depending upon the requested type, either returns the data without modification (bytes) or converts it to an appropriate type (int, float). The Simple Genetic Algorithm fuzzer, centralized and distributed, inserts random bytes during mutation with this method. Conversely, the Mangle algorithm, centralized and distributed, follows a predefined algorithm for attack heuristic generation as defined in Chapter 2 and forgoes requesting attack heuristics from the RFuzz library.

This simplified approach to attack heuristics is a calculated decision. By simplifying attack heuristics, the viability of a genetic algorithm fuzzer can be measured independently of attack heuristic quality. If a genetic algorithm fuzzer is capable of finding defects with randomly generated attack heuristics, then targeted attack heuristics, known troublesome values, should yield additional defects. Since one goal of this research is to improve the metrics available for genetic algorithm fitness functions, improvements in attack heuristics are left as an additional research topic.

4.3 Executor

Executors launch a program under test and define a mechanism for delivery of negative test cases to the running program. The choice of how to run a program and how to deliver a test case is largely fuzzer and application dependent. Some fuzzers require gathering detailed traces of program execution and others only spawn a program. Some applications can process files when specified through the command line and others do not have this capability. Fortunately, Mamba implements two different program execution methods and two different test case delivery mechanisms to placate this variability in applications. These execution methods and delivery mechanisms are completely independent of one another insomuch as an execution method can be combined with either delivery mechanism.

The first implemented program execution method is process spawning. With this method, Mamba forks a process from the currently running fuzzer and replaces it with the program under test. The replacement happens through the Ruby standard library's Process.spawn() method. Once replaced, the framework delivers the negative test case to the program under test and monitors execution of the forked process to determine when to kill it. The timeout parameter, found in the global configuration file (fuzzer.yml), prescribes the amount of time to wait before killing the process. After it is killed, the framework proceeds to the next generated test case, and the same procedure continues until all negative test cases have been tested.

Process spawning forgoes providing any additional information about functions or basic blocks executed during processing of a test case. It likens itself to a black box stimulus and response system. The only available information is whether the program under test exits normally or due to an exception. Exiting due to an exception would be a positive response since a fuzzer's goal is to locate software defects. Process spawning is only suitable for fuzzers without a requirement for dynamic analysis metrics to generate their negative test cases. These fuzzers are typically of the mutation or generation classification. Specifically, within the Mamba framework, the Mangle fuzzer demonstrates this execution method.

The other execution method implemented for Mamba is emulation with Valgrind. Valgrind, a dynamic binary instrumentation framework, supports dynamic binary analysis through an innovative plug-in tool architecture. It starts by loading a program specified by a user, translating the program's instructions into an Intermediate Representation (IR), permitting a plug-in tool to examine and alter the IR, and then translating the resulting IR back into machine code. This translation occurs continuously as basic blocks of a program are encountered. It is similar to how a just-in-time (JIT) compiler interprets and compiles code as it is encountered [Nethercote 2004]. Translating machine

code to an IR, instrumenting the IR, and translating it back to machine code before executing the code is known as a disassemble-and-resynthesize process [Nethercote and Seward 2007].

Valgrind's IR is a Reduced Instruction Set (RISC) language named VEX. The emulator supports conversion of multiple architectures like x86, ppc32, ppc64, and x86_64 into this IR. However of these architectures, only certain platforms are supported. These platforms are Mac OS X and Linux. Other platforms, namely Windows, are currently unavailable, but experimental support is available with the Cygwin project.

Other DBI frameworks concentrate on supporting Windows and Linux while ignoring direct support for Mac OS X. The Pin instrumentation tool only supports x86 Mac OS X applications, and the last software release happened in 2007. Since some applications on Mac OS X 10.6, released in 2009, are x86_64 executables, using the Pin instrumentation tool reduces the set of programs to test on the current version of Mac OS X. The other popular instrumentation tool, DynamoRIO, has not been ported to Mac OS X. Based on this information about Pin and DynamoRIO, Valgrind is the only DBI framework currently suitable for inclusion into Mamba.

To emulate a program, Valgrind translates its version of a basic block between VEX and machine code. For Valgrind, a basic block is defined as a sequence of assembly instructions ending with a control flow altering instruction (e.g. jmp or call)[Nethercote 2004]. This termination of a basic block aligns with IDA Pro's definition of a basic block containing only one exit point of which is a control flow altering instruction. The start of a Valgrind basic block is also comparable to IDA Pro's definition. Valgrind basic blocks start with an assembly instruction that is the target of a control transfer instruction. IDA Pro's distinction of the start of a basic block is often the target of a control flow altering instruction like a branch [Eagle 2008].

However, in some instances, these distinctions of basic blocks diverge making correlation of start and end addresses between Valgrind and IDA Pro infeasible. An example of this divergence happens when Valgrind follows an indirect jump through a stub code section. In many cases, IDA Pro attribute extraction does not export information about indirect jumps, especially through different sections, but Valgrind correctly records control flow changes throughout these sections. An example would be an indirect call through the symbol_stub section of a Mach-O binary. Another example of divergent basic block definitions occurs due to Valgrind's just in time compilation. Valgrind's basic block definition permits code to jump into the middle of a basic. This behavior is permissible, because Valgrind can only calculate the start of a basic block by observing control flow diverting to another set of sequential instructions which may contain a target of another branch or jump elsewhere in the program. While these differences in accounting decrease fidelity of static and dynamic analysis address resolution, the addresses of function prologues can always be matched. Since most metrics

are associated with executing a function and not individual blocks, recording hit tracing information with Valgrind for later comparison with IDA Pro disassembly is possible.

Translation of machine code in Valgrind occurs in an eight phase process. The first phase disassembles machine code into an unoptimized form of VEX IR. Figure 4.3 displays an example translation of an x86 instruction to VEX IR. Note that, unlike Ida Pro, Valgrind's debugging output for basic blocks uses AT&T syntax. The figure shows a movl x86 instruction setting the ebp register to the value stored in the esp register. An instruction commonly found in a function prologue. The corresponding VEX translation of this instruction is three statements, an IMark, PUT, and GET. The IMark statement is purely informational. It specifies the address at which the x86 instruction is found and its length in bytes. The GET and PUT VEX statements read and store data from and to the guest state which is a set of registers stored in memory to emulate program execution. The parameters in these statements are the byte offset of the register in the guest state. For this research, only addresses of executed blocks are of concern since the Mamba framework cross references these addresses with addresses exported from IDA Pro. A more thorough treatment of the VEX instruction set is available in the comments of libvex_ir.h file found within Valgrind's source code.

```
0x8FE01032:  movl %esp,%ebp

    ------- IMark(0x8FE01032, 2) -------
    PUT(60) = 0x8FE01032:I32
    PUT(20) = GET:I32(16)
```

Figure 4.3: VEX IR Example

Valgrind's next phase optimizes the translation before passing the IR to the third phase for instrumentation. During the instrumentation phase, a tool can alter the IR in any way before relinquishing control. For the Mamba fuzzing framework, since tracing the addresses of blocks executed is of concern, a new instrumentation tool was developed named Rufus. This tool passively monitors and records program execution in XML format so that Mamba can parse the trace after execution ceases. Figure 4.4 illustrates an example trace recorded by Rufus of an application using the vulnerable TLV parsing library. Notice that the address of the vulnerable_parser() function, 0xe40, matches the start address exported from IDA Pro (Figure 3.3).

To make these addresses match, the Rufus Valgrind tool must normalize recorded addresses by subtracting the base address of the shared library. This base address is the address in memory where the start of the library is loaded. Normalization is required partially because of Mac OS X address space layout randomization and partially due to Valgrind's control of these addresses.

```
<status>
  <state>RUNNING</state>
    <time>00:00:00:00.037 </time>
</status>

<hit><funcname>vulnerable_parser</funcname><offset>0xe40</offset></hit>
<hit><funcname>vulnerable_parser+336</funcname><offset>0xf90</offset></hit>
<hit><funcname> stub helpers+12</funcname><offset>0xfa2</offset></hit>
<hit><funcname>markov:-[NSConcreteFileHandle offsetInFile]</funcname><offset>0↩
    x6543ba</offset></hit>
```

Figure 4.4: Vulnerable Library Trace

When a program is loaded for emulation, Valgrind sets the DYLD_SHARED_REGION environment variable to the keyword avoid. By setting this environment variable, it directs the Mac OS X loader, dyld, to ignore the shared cache files when determining the address to load a shared library in memory. This effectively nullifies address space layout randomization. However, Valgrind still may arbitrarily load a shared library at different memory locations on different machines. Since IDA Pro exported addresses are based on a displacement from zero, the Rufus Valgrind tool queries the Valgrind core for a shared library's start address with the VG_(DebugInfo_get_text_avma)() function. The returned address is subtracted from all recorded addresses. Normalization guarantees address resolution independent of Valgrind runs and even independent of the machines running the emulation.

The Rufus Valgrind tool also minimizes which addresses are recorded during execution by only monitoring one shared library. The library to monitor is set by the −−object command line option when starting Valgrind with −−tool=rufus. When setting this option, it is important to specify the full path of the shared library since Rufus performs a string comparison of this value against the full path used by the linker. The Mamba framework hides setting this option and determining full path information from the tester. The shared library is parsed from the IDA Pro exported information (Figure 3.3) disName field.

After a Valgrind tool has instrumented the IR control passes to phase four. This phase optimizes the freshly altered IR by removing dead code. Optimized IR then passes to phase five where it is converted into a tree structure. The tree structure passes to phase six where the statements are converted to assembly instructions using virtual registers. Phase seven allocates actual registers based on the instructions by using a linear-scan register allocator. The last phase then converts the assembly instructions into machine code and executes it [Nethercote and Seward 2007].

Utilizing Valgrind for dynamic analysis, in this case hit tracing, is expensive in terms of execution time. Research has shown a slow down of as little as 4 times normal execution time to 22 times normal

execution time depending upon the Valgrind instrumentation tool. The decision to use emulation, in spite of this slow down, is based on inconsistent support of other hit tracing tools. Most other methods use low level system calls to create automated debugger interfaces. Two notable examples of this technique are the Ruby Ragweed module and PaiMei's Pydbg. Each of these require setting breakpoints for all basic blocks before executing an application. When an application hits one of the breakpoints, then a script can record it. However, relying on wrapping a debugger complicates hit tracing since address space layout randomization changes shared library starting addresses. This situation can be complicated even more if symbol tables are removed from an executable or shared library. On Mac OS X, the shared cache files can be parsed to determine randomized addresses, but initial testing with Ragweed setting over a thousand breakpoints caused the Ruby interpreter to stall. By emulating a program, polling for the shared library start address, and subtracting this address from each recorded address to create a displacement from zero, Mamba can scale in a distributed environment as shown in Chapter 6 to combat the increase in execution time due to emulation.

To complement process spawning and emulation, Mamba provides two negative test case delivery mechanisms. The first is through Mac OS X's Applescript language. Applescript is a scripting language that allows the control of common application tasks. Applescript, like Ruby, is object oriented, but scripts are compiled before execution instead of interpreted. The Automator utility installed with the Mac OS X operating system presents a graphical user interface that abstracts programming with Applescript so users can write scripts without learning the language.

Fortunately, a Ruby library exists, appscript, to bridge Ruby and Applescript. This bridge allows a Ruby script to send commands directly to a running application to perform tasks such as opening a file. The opener.rb script, installed with the Mamba framework, uses this library for that purpose. When the executor in the global configuration file, fuzzer.yml, is set to the value appscript, then, when a test case is generated by a fuzzer, Mamba calls opener.rb with command line arguments specifying the process id of the program to test and the file for the program to open. The opener.rb script uses Applescript to obtain a reference to the program, and then, it calls the open() function with that reference and the test case filename. This function call causes a program scriptable by Applescript to process a negative test case.

Not all programs are scriptable through Applescript. The other delivery mechanism handles this deficiency by opening files by command line arguments. This functionality should be useful on other platforms like Linux where Applescript does not exist. To use the command line delivery mechanism, the executor must be set to 'cli#'. After the hash, any command line option can be set. For example, a valid executor configuration could be 'cli#-l -z -f'. When Mamba delivers a negative test case to the program under test, it will remove the 'cli#', spawn a process with the options -l -z

-f, and append the test case filename to the end of the parameter string. This feature is currently experimental, and most testing to date has focused on test case delivery by Applescript.

4.4 Monitor

Program monitoring observes the execution of a program under test to evaluate the results of running a negative test case. Traditionally, methods for monitoring are tracing the program under test with a debugger, scriptable debugger interface, or by monitoring all the signals bound for the program being tested. Since Mamba implements two disparate execution methods, processing spawning and emulation, a different methodology for program monitoring is required. For this, Mamba monitors filesystem activity of directories where crash log information is stored.

For Mac OS X, when a program crashes, the operating system creates a file in the directory `~/Library/Logs/DiagnosticReports` and symlinks another file in `~/Library/Logs/CrashReporter` to the file created in the DiagnosticReports directory. This newly created file provides a traceback of the crash, state of all the crashed process threads, and state of the CPU registers at the time of the crash. By monitoring either of these directories for the creation of a new file, the framework can determine if a test case causes a crash. A Ruby library, Directory Watcher, can monitor file system activity on most operating systems supporting Ruby. The usage of this library alleviates the need for platform specific filesystem monitoring libraries like Linux's inotify or Mac OS X's FSEvents API.

However, a crash occurring during emulation does not cause the operating system to create a crash log. Instead, the emulator, Valgrind, handles the fault, prints descriptive information about the crash, and exits emulation. Only crashes of the emulator itself cause Mac OS X to create a crash log. Emulator crashes typically correspond to an unsupported assembly instruction for Valgrind, and, in rarer situations, an actual error due to a defect in the emulated program. In order to unify monitoring for both executors, the Rufus Valgrind tool tracks faults by registering a fault catcher with the Valgrind core. When Valgrind encounters a fault, it calls the registered fault catcher, track_faults(), which in turn creates a file in the directory being watched by Mamba. This function then populates the file storing trace information with a stack trace of the currently executing thread. Figure 4.5 displays a snippet from an example stack trace collected during a test of an executable using the vulnerable TLV parsing library.

The stack trace is printed in XML with the tag fault denoting the start of a detected crash. The fault tag encloses a signal tag with the signal sent to the emulated program and multiple frame tags collected from the Valgrind core. Each frame tag contains an ip tag specifying an instruction address, an obj tag denoting the shared library where the address resides, and an fn tag with the

name of the function containing the assembly instruction at the ip address. The first frame in the fault is the location of the uncovered defect. In Figure 4.5, a defect has occurred with the instruction at address 0xFFFF088C. Since Valgrind is unable to resolve the function name for this address, the fault is more likely in the function before this instruction. Examining the stack trace, the function before this one is the vulnerable_parser() function in the TLV parser shared library which contains a stack buffer overflow.

```
<fault><signal>11</signal></fault>  <stack>
    <frame>
      <ip>0xFFFF088C</ip>
    </frame>
    <frame>
      <ip>0x12F54</ip>
      <obj>/Users/rseagle/Desktop/dissertation/sharedLibrary/tlvparser.dylib</obj>
      <fn>vulnerable_parser</fn>
    </frame>
  </stack>
```

Figure 4.5: Rufus Fault Detection

Normally, a monitor only watches for errors resulting from executing a negative test case. However, with file format fuzzing, a program may execute a test case forever without exiting. To handle this situation, two different techniques can be implemented. The first technique is to time program execution and kill a process after a predetermined threshold. This technique is implemented by Mamba and illustrated in Figure 4.1 where the timeout value is set to five. Mamba uses this timeout value to determine when to kill a program.

While a predetermined timeout value guarantees uniform execution time of test cases, this restriction can cause a fuzzer to miss defects. A program may not process all test cases equally and additional parsing may yield an error. Due to this parsing inequality, a second technique based on CPU monitoring is required. Mamba permits the timeout value in the general fuzzer configuration file to be set to zero. When it is set to zero, the framework monitors CPU utilization of the process executing a test case. When utilization drops below five percent or a test case runs for more than sixty seconds, then Mamba kills the process. An upper bound of sixty seconds guarantees a test case does not run forever, but it still permits an application to parse a test case for an extended period of time. It is recommended to always use CPU monitoring when testing applications, unless testing must be done within a certain time period.

Implementing multiple executors and monitors can lead to unmanageable code. Allowing each executor to be mixed with each monitoring technique can cause spaghetti code with multiple conditionals checking global configuration parameters during the execution of each test case. These

conditionals can significantly impact runtime performance and code structure. Mamba addresses this issue by employing metaprogramming and dynamically writing executor methods when a fuzzer starts. To achieve this flexibility, Mamba checks the global executor parameter on startup to deduce how test cases will be delivered to a program under test. Based on this value, the executor class dynamically defines a run() and valgrind() method for test case delivery via CLI or Applescript with the define_method() Ruby Module function. The run() method handles process spawning and the valgrind() method handles emulation.

Within these dynamically defined methods, a monitoring function is called after the program under test is spawned. Monitoring function selection occurs based on the value of the timeout global configuration parameter. If the timeout value is set to zero, then the executor calls the cpu_scale() method to monitor process utilization. Otherwise, the executor calls cpu_sleep() which sleeps for the time period specified by the timeout parameter. The call to either method is dynamically defined in the run() and valgrind() methods during fuzzer initialization. By using metaprogramming to dynamically define these methods, Mamba factors out conditionals and simplifies overall code structure.

4.5 Logger

Logging is arguably one of the most important components of a fuzzing framework. A logger records the progress of a fuzzer, details surrounding negative test case generation, and when a program executes a test. Loggers can even record detailed information about attack heuristics inserted into a test case. The more detailed information a fuzzer logs during testing, the easier it becomes to reconstruct a testing scenario.

For Mamba, the logging requirement is satisfied by the usage of the Ruby Log4r library. It is an open source logging library providing different log levels (DEBUG, INFO, WARN, ERROR, FATAL), various output schemes (file, stdout, email, etc.) and remote logging capabilities. The Mamba framework uses Log4r to record information about fuzzer runs to files in the logs directory of a fuzzing environment. Files under this directory are named 'type.YYYY-MM-DD-HH-MM-SS.log'. The type corresponds to the type in the global configuration file, and the date and time reflect when the fuzzer started. Each time a fuzzer is started, a new log file is created to hold only information about that run.

Mamba permits fuzzers to use this logging mechanism in any method suitable for its algorithm. During initial setup in the base fuzzer class, before yielding execution to fuzzer specific code, the framework initializes an instance of the logging class and creates a logging file. This instance is

inherited by all fuzzer specific code using the base fuzzer class as its parent. Any fuzzer can then log any messages to the log file by using Log4r's API.

When logging information about fuzzing, the minimum should be current fuzzer configuration, the time when a program runs a test, and statistics about the job after it concludes. These concluding statistics should present a summary of the number of crashes found, the test cases causing crashes, the total elapsed time for the job, and the total number of tests run. Other information about a particular algorithm is also helpful for debugging. All of these message must be prepended by a date and time, which is automatically handled by Log4r. By systematically recording detailed information about fuzzing runs, the accuracy and speed of crash validation and analysis can be improved.

4.6 Reporter

The distinction between a logger and a reporter is subtle. A logger continuously records information about a fuzzing run as it is occurring, while a reporter polls the environment on demand for summary statistics. The polling presents an immediate update on interesting statistics like the crashes found to date, elapsed run time, or the number of tests executed up to this point in time. Reports provide snapshots about the current fuzzer status.

Mamba implements a reporter by defining a thor task. This task, '`thor fuzz:report`', sends a SIGINFO signal to the currently running instance of Mamba. When received, Mamba's reporter class catches the signal and prints summary statistics about the currently running fuzzer. These statistics are the job start time, elapsed run time, number of test cases run, number of crashes found, and the files that likely caused the crashes. All of this information is written to a file in the fuzzer environment's logs directory, 'Reporter-YYYY-MM-DD-HH-MM-SS.log'. The date and time within the filename reflect the fuzzer's start time. It is important to note that test cases listed in the report as causing a crash are only suspects, because the crash monitoring polls the filesystem on five second periodic intervals. It is possible for a test case to cause a crash and another test case to begin executing before the monitor polls the filesystem. However, this can be resolved by examining the timestamps in the logs and cross referencing timestamps on core files.

This chapter discussed the theory and implementation behind the Mamba fuzzing framework. It walked through creation of a fuzzing environment, configuration of the Mangle fuzzer, and described the software architecture of Mamba. This framework overview solidified the classification strategy presented in Chapter 2, so that further chapters can build upon it by adding search algorithms, namely genetic algorithms.

Chapter 5

Genetic Algorithms

Previous chapters, presented a foundation for fuzzing programs with genetic algorithms. Chapter 3 walked through binary analysis of Mac OS X programs and described a disassembly attribute extraction process. Chapter 4 created a fuzzing framework especially designed for genetic algorithm fuzzing, Mamba, and detailed its components and their relationships to a fuzzer taxonomy given in Chapter 2. This chapter unifies previous concepts and processes by extending the Mamba fuzzing framework with multiple genetic algorithms. This extension consults information gathered during static and dynamic analysis to shape evolution and leverages Mamba's configuration scheme, attack heuristics, emulation executor, monitoring, logging, and reporting for typical fuzzer behavior.

Before delving into implementation specifics, this chapter first introduces some terms related to genetic algorithms. After defining terminology, the next section describes Mamba's genetic algorithm implementations while illustrating how to configure Mamba for genetic algorithm fuzzing. Following that section, this chapter outlines a set of variables to create mathematical equations for fitness functions based on static and dynamic analysis. The chapter concludes with a comparison of Mamba's genetic algorithms against two popular fuzzing algorithms, Peach and Mangle.

5.1 Foundations

John Holland pioneered genetic algorithms, a subcategory of what is known today as evolutionary computation, at the University of Michigan during the 1960's and 1970's. Holland was fascinated by how an organism in nature evolved to progressively improve its ability to survive in an environment. He questioned if and how this environmental adaptation could be applied to other domains outside of Biology. Through a comparative study of multiple fields (Genetics, Game Theory, Economic Planning, etc.), Holland determined that evolution, specifically adaptation, could be directly applied

to other domains by defining a formal framework to model adaptive systems. Central to this formalization was the notion of using a performance measure of an individual within a population, to model natural selection and survival of the fittest [Holland 1992].

For Holland to solidify his framework, he adjusted multiple concepts from Biology and Genetics to conform to a simplified mathematical model of evolution. Instead of defining these concepts without perspective, let's first define an example problem and terminology while walking through an application of a genetic algorithm to solve this problem. For this example, recall the problem of maximizing a quadratic equation, specifically the one show as Equation 5.1.

$$f(x) = -x^2 + 4 \tag{5.1}$$

A direct method for finding the maximum value of $f(x)$ is to take the first derivative of $f(x)$ and solve for zero. Since the degree of the polynomial is two and the coefficient of the squared term is negative, then the parabola defined by the function opens downward. This means the parabola's vertex must be the function's maximum value, and solving the first derivative of the equation for zero produces the parabola's vertex.

For this example, let's now constrain the permissible methods for finding the x value maximizing the equation to only direct function evaluation. One might then proceed to enumerate or randomly select a number from the set of integers ($\mathbb{Z}$) in an attempt to locate the x value that maximizes the function.

Another approach is to search for a solution by applying a genetic algorithm. A genetic algorithm can measure an x value's performance, otherwise known as fitness, by evaluating Equation 5.1. A higher value returned by the equation denotes an x value more suitable for maximizing the function. This measurement of a candidate solution based on its performance is the same concept Holland borrowed from Darwin to promote evolutionary adaptation in a mathematical model.

How do genetic algorithms represent a candidate solution, and, how do these candidate solutions evolve over time? For the answer to these questions, let's delve deeper into the composition of a candidate solution. A candidate solution is referred to as a chromosome, and chromosomes have associated encodings. A chromosome's encoding is a matter of choice, but the most common encoding is a binary representation where a chromosome is a sequence of zeros and ones. For our example problem, and all genetic algorithms created for this dissertation, chromosomes conform to a binary encoding.

Chromosomes are also divided into smaller building blocks known as genes. For a binary encoding, a gene is an adjacent subsequence of zeros and ones. The length of this sequence can range from a single zero or one to the total size of the chromosome. These genes encode different

traits of a chromosome [Mitchell 1998]. For instance, a chromosome could encode two one byte integers, because a function being optimized requires values for two variables. A chromosome for this problem could then be composed of two genes with each gene being a byte in length. Each of the genes are values for the function's variables, and these values are traits of the chromosome. For our example problem, let's define a chromosome as a one byte entity with the genes being one bit in length. Each gene encodes a trait, namely the value of a bit of an 8-bit signed integer composing the x variable for $f(x)$. This encoding and composition allows a genetic algorithm to evolve individual bits of an integer's base two representation. When a chromosome is evaluated for fitness, it will be decoded into an eight bit signed integer, and Equation 5.1 will be evaluated for the chromosome's fitness.

Genes also have a range of possible values and a position within the chromosome. The range of values is referred to as a gene's alleles. The different locations of genes in a chromosome are the chromosome's loci. Alleles and loci are convenient terms to use when discussing the reproductive process of chromosomes and genetic operators [Mitchell 1998]. Since our example defines genes as one bit, the alleles of these genes are the values zero and one. The loci of the chromosomes are each bit position of the one byte signed integer representation.

At this point, we have defined a genetic algorithm to represent and measure the fitness of binary encoded candidate solutions. However, encoding, decoding, and evaluating a function for fitness may seem synonymous with an enumeration or random search algorithm. How do genetic algorithms differentiate themselves from these other algorithms? The answer is by evaluating a group of candidate solutions and evolving the group over time with genetic operators. A group of candidate solutions, chromosomes, is known as a population, and a population has an associated size. This size dictates the number of chromosomes that must be in the population at any particular time [Sivanandam and Deepa 2007b].

A special designation is given to the population initializing a genetic algorithm. This population is referred to as the initial population. These populations may be randomly generated or hand selected by a researcher to possibly decrease the computational time required to converge to a "good" candidate solution. An initial population fulfills an important role in evolution. Hand selected initial populations may maximize differences between chromosomes or start with a set of chromosomes with particularly high fitness. For our example, let's define the population size to be four and the initial population to be $G_0 = \{1, 5, 2, 6\}$.

Since populations evolve over time, a term must be created to differentiate populations at particular points in time. This term is a generation, and in some literature on genetic algorithms, they refer to genetic algorithms as generational search techniques, because a genetic algorithm evaluates groups of possible solutions over time. More formally, a generation is a population at

a given time before genetic operators are applied to create the next population. In our case, G_0 is the first generation. After all chromosomes in the current generation are evaluated and new chromosomes are bred based on the older chromosomes, then the resulting population becomes the next generation, G_1. A genetic algorithm halts producing generations when either a predefined number of generations has been reached or when a particularly "good" solution is found. These criteria are not the only valid stopping conditions, but, in practice, the former usually serves as a useful boundary.

The only undefined components of a genetic algorithm left are its reproductive process and genetic operators. For reproduction, Holland suggested four operators: selection, crossover (recombination), mutation, and inversion [Holland 1992]. Of these operators, the inversion operator is seldom used, and is not defined or used by genetic algorithms developed for this dissertation. Figure 5.1 abstractly illustrates the other genetic operators, namely selection, crossover, and mutation.

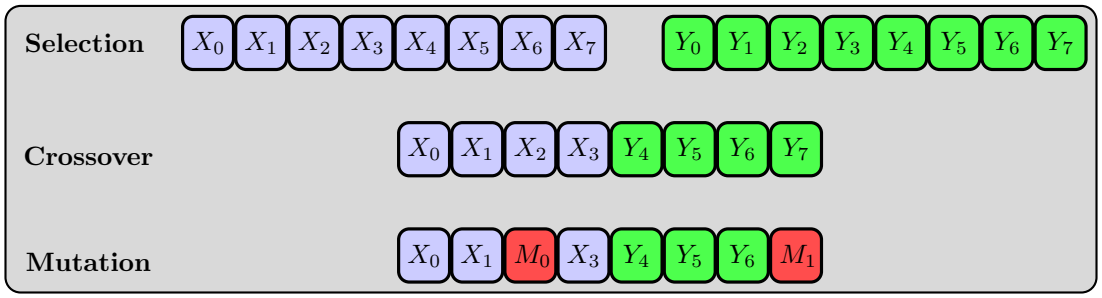


Figure 5.1: Illustration of Selection, Crossover, and Mutation

Reproduction occurs after all of the chromosomes in a population have been evaluated by a fitness function being optimized. After evaluation, all chromosomes have fitness metrics associated with them promoting how well a chromosome solved the problem. The selection operator now chooses parent chromosomes usually with replacement based on their fitness metrics. Replacement means a chromosome may be selected as a parent multiple times. Many selection schemes exist, and, in most schemes, the selection operator strives to choose the most fit chromosomes from a population to breed. By choosing fitter chromosomes, a genetic algorithm can model survival of the fittest [Mitchell 1998].

Some of the common selection schemes for a genetic algorithm are roulette wheel selection, rank selection, and tournament selection. Roulette wheel selection, a form of fitness proportionate selection, weights the probability of selecting a chromosome by its proportion of fitness out of the entire population's fitness [Mitchell 1998]. Many texts analogize a roulette wheel where the size of slots on the wheel are derived from a chromosome's proportional fitness to roulette wheel section. This selection method is implemented for most genetic algorithms implemented for this

dissertation. Rank selection normalizes proportionate selection by numbering chromosomes based on their observed fitness values. These ranks begin at one for the worst fit and increment through the number of population members. Higher ranked chromosomes have a higher probability of being selected. By ranking individuals based on fitness, the probability of selecting one particular chromosome can be lessened in situations where that chromosome's fitness accounts for the majority of the population's total fitness [Sivanandam and Deepa 2007b]. Tournament selection randomly picks a smaller subset of chromosomes from the entire population and compares their fitness metrics. The most fit chromosome of the subset is chosen as the winner of the tournament and as a parent for reproduction.

Revisiting our example, the fitnesses of chromosomes in the initial population are 3, -21, 0, and -32. Since the fitness function being optimized (Equation 5.1) can produce negative numbers for fitness metrics of population members, these fitness measures must be remapped to facilitate roulette wheel selection. To remap fitness metrics, we will exploit the fact that an eight bit encoding of signed integers only permits the range of values $[-128, 127]$. Since a genetic algorithm is trying to maximize a quadratic equation, it can remap fitness metrics by adding the absolute value of the fitness function's minimum value for an eight bit signed integer to all fitness metrics. This minimum occurs when the x value of the fitness function is -128. The absolute value of the result returned by $f(x)$ for this x value is 16380. This leads to Equation 5.2 as a remapping function for $f(x)$.

$$g(x) = f(x) + 16380 \tag{5.2}$$

Equation 5.2 guarantees fitness values to be greater than or equal to zero while preserving the order of chromosomes based on fitness calculated from Equation 5.1. With this remapping function, the fitness values of the example's initial population are $16,383$, $16,359$, $16,380$, and $16,348$. The sum of fitness metrics constituting the overall initial population's fitness is $65,470$. Expected values for probability of selection for each chromosome can now be assigned to each chromosome by dividing their fitness derived by $g(x)$ by $65,470$.

Reproduction selects chromosomes as parents for as many times as required to breed another population of identical size. Selected parents are passed, often in pairs, to the next genetic operator, crossover. This operator combines genes from the selected parents by uniformly choosing a single loci of the parents at random. The genes above the loci of the first parent are copied and the genes below the same loci of the second parent are copied to create an intermediate child chromosome. For optimization purposes, the sections of genes copied from both parent chromosomes are switched to create a second child chromosome. This optimization reduces the number of selections from the population. The scheme presented here is known as single point crossover, because a single loci is

used as a crossover point. Other crossover schemes exist such as two point crossover, multipoint crossover, uniform crossover, and many others [Sivanandam and Deepa 2007b].

Crossover is also governed by a probability of crossover (P_C). Crossover probability defines the probability of applying the crossover operator to the selected parents. This probability ranges from 0% to 100%. In the case where crossover is not applied to two parents, the parents are passed unchanged to the next genetic operator, mutation, otherwise the mutation operator alters child chromosomes. Figure 5.1 illustrates an intermediate child chromosome resulting from single point crossover of parent chromosomes with eight genes ($X_0, X_1, \ldots, X_7$ and $Y_0, Y_1, \ldots, Y_7$). The crossover point selected occurs at gene five and the resulting intermediate child chromosome is shown. For most genetic algorithms implemented for this dissertation, single point crossover serves as the crossover operator. The only exception is the CHC-GA which is described in more detail in Section 5.2.

The last genetic operator, mutation, selects genes of the intermediate children of parent chromosomes, and it randomly changes the genes value. Like crossover, mutation is governed by a probability of mutation (P_M). However, this probability dictates the probability of mutating a gene in the chromosome. This means each gene in a chromosome has a probability of P_M of being mutated. In Figure 5.1, mutation alters gene X_2 to M_0 and gene Y_7 to M_1. The value of the mutation must be a possible value of the gene's alleles.

To tie these concepts back to our example, let's select two highly fit individuals from our initial population. For this, we select the chromosomes with decoded signed values 1 and 2 as parent one and parent two. The speculation here is that roulette wheel selection caused these two parents to be chosen. The eight bit binary representation of these chromosomes are 00000001 and 00000010, respectively. If single point crossover is followed as in Figure 5.1 and a crossover point at gene five is chosen, then the intermediate child chromosome would be 00000010. Continuing through the reproductive process and matching the figure's depiction, applying mutation creates a child chromosome of 00100011 equaling 35. This child chromosome is then placed into the population of the next generation, and more children are bred to reach the next population's required size.

One last concept that can be performed after all child chromosomes are created for the next generation is a concept known as elitism. Elitism selects the best n chromosomes from the current generation and replaces n chromosomes of the next generation with these selected chromosomes. The n, in this case, is a number significantly less than the total population size. By selecting the fittest chromosomes of the current generation and promoting their survival in the next generation, a genetic algorithm preserves highly fit chromosomes which may degrade in fitness through crossover and mutation [Mitchell 1998, De Jong 1975]. Every genetic algorithm, except for CHC-GA, currently implemented in the Mamba fuzzing framework uses elitism to preserve highly fit chromosomes.

As shown by our example problem, genetic algorithms can be applied to search space problems. In the example, the search space was the set of eight bit signed integers, a subset of $\mathbb{Z}$, and a genetic algorithm tried to maximize Equation 5.1 by finding the best x value in that search space. While genetic algorithms are helpful for generating candidate solutions to search space problems in these cases, how are these algorithms applicable to file format fuzzing? From a quality assurance perspective, genetic algorithms do not appear applicable to the problem since this perspective presents fuzzing as a methodology for improving software quality by constantly injecting semi-valid data into a program under test and monitoring its responses. However, this formalization provides little perspective of the actual problem a fuzzing algorithm is trying to solve, and it is a classical illustration of approaches taken by mutation and generation fuzzers.

In actuality, fuzzing algorithms are searching for negative test cases in the set of all inputs that cause a program under test to exit abnormally. Vuagnoux [2005] demonstrated this shift of perspective with his Autodafé fuzzer. In these cases, the search space of a program is the set of all possible inputs through any external program interface. For file format fuzzing, this external interface is any file parser. Posing fuzzing in this form reclassifies fuzzing as a search space problem with multiple optimal solutions. A fuzzer's goal in this characterization is now to find as many of these optimal solutions, crashes, as possible in a limited time period defined by a tester. Since genetic algorithms are useful for examining a search space and their operators generate variations of chromosomes, it is natural to extend these algorithms to file format fuzzing with files as chromosomes. Perhaps by optimizing certain fitness functions associated with different program metrics, a genetic algorithm may cause program faults while optimizing a function.

5.2 CHC-GA

Traditional genetic algorithms, those following a strict generational evolutionary model with traditional selection, crossover, and mutation operators do not constitute all possible evolutionary algorithms. Other evolutionary algorithms exist which build upon Darwinian concepts. These other algorithms try to improve evolutionary algorithms with variations of genetic operators. Some of these algorithms also extend breeding competition by allowing children to compete with parents in cross-generational selection schemes. One such genetic algorithm extending the mathematical evolutionary model is the CHC-GA.

CHC-GA differs from traditional genetic algorithms by several facets. In general, CHC-GA heavily weights survival of the fittest during its reproductive cycle. This heavy reliance on survival of the fittest is exhibited by CHC-GA's conservative cross-generational selection operator. For selection, CHC-GA increases its population size to include all current chromosomes as well as all children

yielded from crossover. The resulting children are evaluated for fitness, and then, CHC-GA selects the most fit individuals from this combined, intermediate population to create the next generation. Combining parents and children and selecting from the intermediate population allows CHC-GA to test new chromosomes yet retain the most fit parent chromosomes which may be damaged during a destructive crossover or mutation operator. Eshelman [1990] denotes this selection mechanism as population-elitist selection.

While population-elitist selection strives to conserve high performing chromosomes, the crossover operator for CHC-GA maximally increases genetic differences between parent and child chromosomes. To achieve such a highly disruptive crossover operator, CHC-GA utilizes a variation of Uniform Crossover (UX) deemed Half Uniform Crossover (HUX) [Eshelman 1990]. HUX compares each bit, a gene if the encoding dictates a gene as a bit, of two selected parents, P_1 and P_2, to determine their equivalence. If P_1 is denoted as a template for the resulting child, then exactly half the differing bits from P_2 are chosen to be copied to the resulting child. These chosen bits remain at the same loci in the child as found in P_2, and the remaining bits are collected from P_1 to complete crossover. This highly disruptive crossover operator yields children distinct from either P_1 or P_2.

One concern, noted by Eshelman [1990], when combining a conservative selection strategy with a highly disruptive crossover operator is the tendency for a population to converge prematurely. Premature convergence should be thwarted by crossing over multiple bits to create maximally different children, but situations can arise where "good" chromosomes crossover with one another and their differences are minimal. In some cases, this incest leading to convergence is desirable, but in others, the convergence may be premature, curtailing search.

CHC-GA incorporates an incest avoidance mechanism. This avoidance mechanism occurs during the breeding of children, more specifically during selection. When parents are selected, at random with replacement, the Hamming distance between the two selected chromosomes is calculated. If the Hamming distance between these parents falls below a quarter of a chromosome's length, then the parents are not permitted to mate. This threshold is denoted by CHC-GA as a difference threshold [Eshelman 1990]. The reproduction cycle continually selects parents for mating as many times as the size of the current generation despite failures due to the difference threshold. If a child does not result from selection of two parents because of the incest prevention threshold (difference threshold), then the reproductive cycle does not redo selection for that child. This means the number of children generated by selection and crossover can be less than the current generation's size. The next generation's size will be the same size as the current generation, because the next generation is selected from the combined intermediate population of parents and children.

In the event no children are generated, CHC-GA has detected convergence, the incest threshold decrements by one bit. When the incest prevent threshold reaches zero and no children result from

breeding or no children are selected for the next generation, then CHC-GA introduces a mutation operator. For mutation, the algorithm partially reinitializes the next generation by creating a template of the current fittest chromosome. The template is copied into the next generation and 35% of its bits are randomly flipped. This procedure repeats until a new generation is created with a size equivalent to the starting generation's size minus one. To generate the remaining chromosome, an exact copy of the fittest chromosome is copied to the next generation for preservation (elitism). CHC-GA then resets the incest prevention threshold and continues evolving new generations until this process repeats or a stopping condition is reached. This mutation operator is known as cataclysmic mutation [Eshelman 1990].

CHC-GA, being a more sophisticated evolutionary algorithm, is an interesting algorithm to apply to file format fuzzing. However, it must be adapted to work with file fuzzing, and Mamba's implementation of CHC-GA alters this algorithm in multiple ways to accommodate files as chromosomes. The first modification is a redefinition of a gene. Originally, CHC-GA represented chromosomes as a binary encoding with each bit as a chromosome's gene. For file fuzzing, defining crossover and mutation on a bit boundary is too refined since files are generally several kilobytes to megabytes in size. Mamba's implementation of CHC-GA redefines a chromosome as a file with each byte as a chromosome's gene. Redefining a chromosome's genes as bytes instead of bits can lessen the effects of disruptive genetic operators.

Since a chromosome's genes are defined as bytes, Mamba's CHC-GA modifies the original crossover operator to be compatible with this new representation. The highly disruptive crossover operator (HUX) now compares each byte of two parent chromosomes and exchanges half of the differing bytes. While this adaption matches the new encoding, mixing half of the unique bytes from one parent with another generally corrupts a resulting child file to a state beyond recognition. In mature application code, conditionals should catch these highly corrupted files fairly early during execution. In anticipation of this problem, Mamba's CHC-GA provides a configurable crossover parameter (P_C) allowing a tester to set a percentage of differing bytes to crossover. This flexibility permits a software tester to customize their CHC-GA based on application code maturity.

Another modification to CHC-GA made by Mamba is the semantics of the Hamming distance algorithm used during selection. Traditionally, the Hamming distance between two strings is only defined for strings of equal length. If the strings are unequal in length, then another similarity measure such as the Levenshtein distance is used to compute the edit distance between strings. Computing the edit distance by another measure like the Levenshtein distance can increase runtime since insertions and deletions are also being factored into the similarity measure. Since file fuzzing generates files of different sizes and chromosomes can be several megabytes, the Hamming distance calculation for Mamba's CHC-GA counts the number of bytes different between parents and adds

the difference in size between the parents to the calculation. Altering the semantics of the Hamming distance computation aligns with the original CHC-GA while keeping the comparison operator linear.

Also, since chromosomes are variable length, CHC-GA's difference threshold must be redefined to account for variable sized chromosomes. Originally, CHC-GA initialized the difference threshold as a predefined number of bits and decremented the threshold by one bit when convergence was detected. When the threshold equaled zero, then the algorithm entered cataclysmic mutation. For variable length chromosomes, the difference threshold is defined as a percentage instead of an absolute number of bits. To derive the number of bytes required to differ between parents, the largest chromosome in the population is multiplied by the difference threshold percentage. By defining the threshold as a percentage, the adapted algorithm accounts for variable length at each generation. If CHC-GA detects convergence, then instead of decrementing the threshold by one bit, the difference threshold is divided by two. When the threshold is less than 5%, the algorithm enters cataclysmic mutation. Initially, the difference threshold is set to 25% which requires convergence being detected four times before CHC-GA mutates the population.

Mamba's last modification to CHC-GA is within the cataclysmic mutation operator. Mamba's modified mutation operator continues to mutate the current fittest chromosome, but instead of 35% of its bits, the operator inserts a random sequence of bytes (attack heuristic) at a random offset within the new chromosome. This procedure repeats to generate 35% of the next generation's chromosomes. The remaining 65% of the next generation is reinitialized by randomly choosing with replacement from the initial population and preserving the current fittest chromosome (elitism). By altering CHC-GA in this manner, diversity is introduced into the next generation with well formed chromosomes as well as attack heuristics.

5.3 Configuration

The Mamba fuzzing framework includes four types of genetic algorithms for fuzzing. All of these algorithms, SGA, SGA-MM, BGA, and CHC-GA use the same basic configuration file layout. Minor additions are made to configurations where necessary as will be mentioned with SGA-MM. Some graphs in this dissertation reference a Simple Genetic Algorithm with Mangled Initial Population (SGA-MIP). This genetic algorithm is a special case of the SGA where the initial population was generated independently by the Mangle fuzzer before starting evolution. As such, the SGA-MIP configuration file is the same as SGA.

Before tuning genetic algorithm fuzzing parameters, Mamba must first be configured to run a genetic algorithm fuzzer. To configure Mamba for this, the type field of the global configuration file (configs/fuzzer.yml) must be set to one of the available genetic algorithm types. Matching the above

algorithms to their paramaterized counterparts, these possible values are SimpleGeneticAlgorithm, MangleSimpleGeneticAlgorithm, ByteGeneticAlgorithm, or CHCGeneticAlgorithm. A global and fuzzer specific configuration file supporting genetic algorithm fuzzing can be automatically created with one of these types by the mamba command line tool. To generate a fuzzing environment for one of these types, specify the type with the −−type (-y) option when creating a new fuzzing environment.

Once the global configuration file references a valid genetic algorithm fuzzer, fuzzer specific parameters can be tuned. Depending upon the type chosen, fuzzer specific parameters are found in the configuration files SimpleGeneticAlgorithm.yml, SimpleGeneticAlgorithmMangle.yml, ByteGeneticAlgorithm.yml, or CHCGeneticAlgorithm.yml. Figure 5.2 presents an example configuration for the SGA type.

```
---
Crossover Rate: 0.2
Mutation Rate: 0.5
Maximum Generations: 2
Population Size: 4
Fitness Function: As
Initial Population: tests/testset.zip
Disassembly: disassemblies/libFontParser.dylib.fz
```

Figure 5.2: Mamba Genetic Algorithm Configuration

This configuration file has many tunable parameters that should be familiar from the previous theoretical discussion of genetic algorithms. The first parameter, Crossover Rate (P_C), defines the probability for two parent chromosomes to exchange genes during reproduction. Most genetic algorithms in Mamba implement single point crossover with each gene having an equal probability of selection as a crossover point. This mean P_C does not control the probability of selecting a gene as a crossover point, but it dictates if two selected parents will exchange genes. Chromosomes of these algorithms are variable length which alters the semantics of choosing a crossover point. To handle the special case of two parent chromosomes with unequal lengths exchanging genes, most genetic algorithms in Mamba select a crossover point from the smaller chromosome of the parent chromosomes. This selection scheme ensures the crossover point exists in both parents. For crossover itself (see Figure 5.1), the top half of parent chromosome one up to the crossover point is copied into the top half of child chromosome one and the bottom half of parent chromosome two from the crossover point is copied into the bottom of child chromosome one. The process switches to generate a second child. For the second child, the top half of parent two up to the crossover point is copied

to the top half of child two, and the bottom half of parent one from the crossover point is copied into the bottom half of child two.

The only algorithm diverging from this definition of P_C is CHC-GA. This algorithm uses HUX as its crossover operator. For HUX, typically, half the unique bits between the selected parents are crossed. Mamba redefines P_C for this algorithm to dictate the percentage of unique bits to crossover between parents. This means a P_C value of 0.25 would cause a quarter of the unique bits to be crossed between parents to create a child. No matter the algorithm selected, the bounds for P_C remain constant and are $0 \leq P_C \leq 1$.

The next parameter, Mutation Rate (P_M), sets the probability of mutation during reproduction. The bounds for this value are $0 \leq P_M \leq 1$. For SGA, SGA-MM, and BGA the semantics of the mutation operator differ. SGA defines P_M as the probability of selecting an intermediate child from the next generation to mutate. For example, let's set the number of population members to be 100 and P_M equal to 0.20. Based on these settings, during reproduction, SGA selects 20% of the intermediate child chromosomes to apply mutation. Once the child is selected, mutation occurs by uniformly selecting a random gene from the chromosome. Once a gene is selected from the chromosome, mutation replaces the gene with a randomly generated sequence of bytes. The length of this newly generated sequence of bytes is randomly chosen to be between 0 and 10. This mutation operator serves as the attack heuristics of the algorithm by inserting a random byte sequence into a percentage of chromosomes for the next generation.

The SGA-MM's mutation operator is completely different from SGA, but the algorithm uses P_M in the same manner. This probability again determines the probability of selecting an intermediate chromosome for mutation. Once selected, however, the mutation occurs by applying the Mangle fuzzing algorithm to the chromosome. Mangle randomly selects genes from a configurable range of the chromosome and flips the higher order bits of the selected bytes. The range of bytes selected and starting gene for the range are configurable. These parameters are contained in the fuzzer specific configuration file, MangleSimpleGeneticAlgorithm.yml. For more information on these parameters, please refer to Figure 4.2 in Chapter 4.

BGA diverges from SGA and SGA-MM by defining mutation in terms of a classical binary genetic algorithm [Holland 1992]. With this algorithm, P_M defines the probability of mutating a gene of a chromosome during reproduction. For example, if P_M is set to 0.20 and the number of genes in the chromosome is 100 (i.e. 100 byte file), then 20 randomly selected genes will be mutated throughout the chromosome. Furthermore, every chromosome generated during reproduction will contain mutated genes with the same probability. The mutation operator, in this case, randomly generates a single byte and replaces the selected byte with the randomly generated one. These randomly mutated bytes scattered throughout the chromosome are BGA's attack heuristics. For

all genetic algorithms implemented in Mamba, a chromosome is considered to be a variable length negative test case file and a gene is a byte in the file.

CHC-GA does not include a mutation rate in its configuration file. This parameter is missing because CHC-GA follows a set algorithm for cataclysmic mutation. The algorithm dictates that the fittest chromosome of the current generation is copied into the next generation, 35% of the remaining next generation's chromosomes are created by inserting attack heuristics randomly into the current fittest chromosome. The remaining chromosomes left to be generated for the next generation are randomly selected with replacement from the initial population. This deviation from the original CHC-GA adapts the algorithm to file format fuzzing and increases population diversity. The diversity is increased, because most members of the new generation are not based on the fittest chromosome of the current generation.

The next parameter, Maximum Generations, sets an upper bound for the number of generations a genetic algorithm fuzzer will create. The upper bound serves as a reliable stopping condition for evolution. After that parameter, Population Size, configures the total number of chromosomes in a population. Population size remains constant for each generation created during evolution. A parameter coupled with Population Size is the Initial Population. This parameter sets the location of a zip file containing a genetic algorithm's initial population. Since these fuzzers are file fuzzers and chromosomes are files, the zip file must contain the same number of files as set by the Population Size. To collect an initial population of files, Mamba includes a tool, '`thor tools:seed`', that queries Google for files with a specified type (i.e. pdf, jpg, gif, xls). This tool randomly chooses the search results returned, parses them, and then downloads the requested number of documents making sure each document downloaded is unique. Figure C.5 provides additional usage details for this tool within the fuzzing environment.

The Fitness Function parameter is one of the most important parameters for genetic algorithm fuzzing. It defines the mathematical function which a genetic algorithm will try to optimize. This function is a valid mathematical expression in infix notation, and it can include one or more predefined variables. The next section provides a comprehensive discussion of valid fitness function syntax. For any provided function, a genetic algorithm will optimize it by maximizing the function's return value.

The last parameter, Disassembly, configures the YAML file to read for disassembly attribute information. These files are generated by the Mamba tool, '`thor tools:disassemble`' which uses an IDAPython script distributed with the Plympton Ruby gem and IDA Pro. Please refer to Chapter 3 and Figure 3.3 for more information on disassembly attribute extraction. Once all of these parameters are configured with valid values, a tester starts a genetic algorithm fuzzer with the '`thor fuzz:start`' task.

5.4 Fitness Function Variables

At this point, the process for genetic algorithm fuzzing has been stated. This process is to select a program to fuzz, select a shared library used by the selected program for execution monitoring, disassemble the selected library with IDA Pro, extract disassembly attribute information with Plympton's IdaPython script, create a genetic algorithm fuzzer with Mamba, and configure the selected genetic algorithm's parameters. Tools are included within a Mamba fuzzing environment to help with these steps. The missing puzzle piece now is a set of variables representing disassembly attributes and the connection of these variables to deriving fitness functions. The Mamba fuzzing framework includes a set of variables based on disassembly attribute information and application execution. These variables are split into two classes, modifiable and unmodifiable. Figure 5.3 presents these variables along with brief semantics and modifiers.

Variable	Semantics
A	Number of Function Arguments
B	Number of Function Local Variables
C	Size of Function Arguments
D	Size of Function Local Variables
E	Number of Assembly Instructions per Function
F	Function Stack Frame Size
G	Function Cyclomatic Complexity

Variable	Semantics
R	Test Case Execution Time
U	Execution Path Uniqueness
V	Function Coverage

Modifier	Semantics
a	Average
s	Sum

Figure 5.3: Fitness Function Variables

The main division of this variable set is the division between modifiable and unmodifiable variables. Modifiable variables are variables with a meaning when appending a valid modifier, average (a) or sum (s), to the variable. When these variable are included in a fitness function, Mamba parses the execution trace of a negative test case to find all functions executed in a traced shared library. Once collected, Mamba gathers the appropriate metrics from the attribute extraction information. Then, depending on the modifier added to the variable, the gathered metrics for each function are either averaged or summed. If a function occurs multiple times in the trace, then its metrics are included multiple times in the variable's calculation.

Modifiable variables are denoted by the set $M = \{A, B, C, D, E, F, G\}$. To modify one of these variables, the letter a or s should be appended to a variable to attain the desired semantics. For instance, Aa, calculates the average number of arguments for all functions executed. For completeness, the set of unmodifiable variables is the set $Z = \{R, U, V\}$. These variables can

not have a modifier added to them, because semantically the average and sum of these variables is meaningless. The following subsections describe semantics of each variable in more detail.

Number of Function Arguments (A)

Variable A represents the number of parameters passed to a called function. The number of parameters passed is calculated on a per function basis and stored in the YAML attribute information by the IDAPython script, func-auto.py. This script is an adaptation of the PaiMei pida library included in the Plympton Ruby gem developed by this dissertation [Amini 2006]. The func-auto.py script is used by the Mamba task '`thor tools:disassemble`'.

During attribute extraction, func-auto.py walks through every function defined in a disassembled shared library's text section and queries IDA Pro for each function's stack frame and stack frame size. While stack frames are a convention of executing code, IDA Pro can determine the structure of a function's stack frame by analyzing alterations of the current stack frame pointer register, esp, prior to calling a function and during a function's prologue, epilogue, and function execution. By evaluating these alterations to esp, IDA Pro can reconstruct the number and location of a function's local variables and arguments [Eagle 2008].

For the IDAPython script to obtain these metrics, it first accesses a reference to a function's stack frame represented in IDA Pro by calling the get_frame() API function and the frame's size by the get_frame_size() API function. Once obtained, the script walks through IDA Pro's frame structure calculating the offsets of each member in the structure. The stack frame offsets of each member are received by examining the soff and eoff attributes of each structure member. Since in x86 assembly stack frames grow from high memory addresses to low memory addresses, any access to a memory address higher than the saved stack base pointer, ebp, and saved return address designate a function parameter. IDA Pro names these structure members s and r respectively. When the script encounters a reference to memory lower than the saved instruction pointer, it adds it to the number of function parameters and continues processing. Once all the frame structure members have been evaluated for their position in the stack frame, then the total number of function parameters are calculated. This calculation of function arguments aligns with the x86 cdecl calling convention where a function caller is responsible for pushing parameters on the stack and cleaning up those arguments after return from the callee.

An interesting side note is the behavior of the x86_64 ABI. With x86_64 some function arguments are passed through registers (rdi, rsi, rdx, rcx, r8, r9) instead of pushing arguments onto the stack before calling the function [Matz et al. 2010a]. By passing these arguments through registers, the current IDAPython script is incapable of correctly calculating the number of function arguments for

x86_64 objects. Only arguments passed on the stack after the registers are filled will be counted. Care must be taken when planning a fuzzing run to ensure accuracy of this variable, and this variable should only be used, at this time, when testing 32 bit applications.

Number of Function Local Variables (B)

Variable B denotes the number of local variables allocated for a function. As with the number of function arguments, IDA Pro automatically calculates this number by analyzing the stack pointer register, esp, while disassembling a function [Eagle 2008]. The included IDAPython script ascertains this information while calculating the number of function arguments. However, due to IDA Pro's analysis algorithm, the number of local variables reported by disassembly attribute extraction may not equal the actual number of variables declared in the original source code. This discrepancy is based on IDA Pro incrementing the number of local variables when a read or write happens to any address within a function's stack frame. This discrepancy is noticeable for example when unused variables are allocated in source code. Since IDA Pro does not have direct knowledge of source code structure, it must assume accesses to stack addresses are the only local variables. If desired, this discrepancy can be corrected by manually analyzing a disassembly in IDA Pro before extracting attributes with the '`thor tool:disassemble`' tool.

Size of Function Arguments (C)

Variable C holds the size in bytes of all parameters passed to a function. This size in bytes is calculated by the IDAPython script, func-auto.py, during attribute extraction. It calculates this value by calling the get_member_size() IDA Pro API function for every function parameter encountered while calculating the number of function arguments and local variables. The size calculated by this function is dependent upon IDA Pro's auto analysis which only recognizes function arguments if they are accessed by an assembly instruction within a function. If an instruction does not reference the memory address of a function parameter, then IDA Pro and the attribute extraction script is unable to include the size of that parameter in the calculation for variable C. These discrepancies can be resolved by manually marking function arguments after IDA Pro's auto analysis concludes [Eagle 2008].

Size of Function Local Variables (D)

Variable D represents the size in bytes of a function's local variables. This size is the space allocated in bytes by a function's prologue. The prologue for x86 cdecl functions typically subtracts a fixed number of bytes from the current stack frame pointer, esp, within the first few assembly instructions

of a function. The most commonly recognized instruction for this behavior is a sub esp in the prologue. This subtraction allocates space for stack variables and may include padding created by the compiler for stack frame alignment.

Number of Assembly Instructions per Function (E)

Variable E represents the number of assembly instructions within a function. This number is calculated by traversing IDA Pro's representation of a function. The IDAPython script, func-auto.py, first traverses a function's chunk list, then traverses the block list contained within each chunk. The blocks in this list contain the count of assembly instructions composing the block. The sum of all these instructions yields the sum of all assembly instructions for a function. A potential pitfall encountered early in this research is to naively calculate the number of instruction by subtracting a function's start address from its end address. That calculation assumes assembly instructions are fixed length which is not the case for x86 instructions. The reworked attribute extraction script avoids this naive solution and correctly calculates this variable.

Function Frame Size (F)

Variable F represents the total size of a function's stack frame in bytes. This variable is calculated by the IDAPython script, func-auto.py, by querying IDA Pro with the get_frame_size() API during analysis of other function attributes. This API function returns the size in bytes including all local variables, saved registers (ebp, eip), saved return address, and size of function arguments [Eagle 2008].

Function Cyclomatic Complexity (G)

Variable G corresponds to the cyclomatic complexity of a function. Cyclomatic complexity is related to the number of linearly independent paths within a program. These paths are created by decision points altering control flow. For instance, an if statement causes a decision point where code can branch in two divergent directions dependent on the result of the if statement's conditional. While traditionally cyclomatic complexity is a metric for a software program, it can also be applied to individual pieces of a program like functions.

Thomas McCabe developed cyclomatic complexity as a software metric in 1976 [McCabe 1976]. To calculate this metric a program, module, or function is converted to a control flow graph. The nodes of the graph correspond to sequential instructions and edges relate to branches to and from non-sequential sections of code. This representation may be familiar from discussions of IDA Pro

disassemblies in Chapter 3. Once converted, the complexity is computed by the formula shown in Equation 5.3.

$$CV = e - n + 2p \tag{5.3}$$

where e is the number of edges, n is the number of nodes, and p is the number of connected components in the control flow graph. A minor simplification of this formula is possible by assuming all components of the graph are connected. Since this research calculates cyclomatic complexity of a function, it is feasible to assume all assembly blocks composing a function are connected. Then, the calculation simplifies to Equation 5.4.

$$CV = e - n + 2 \tag{5.4}$$

The IDAPython script, func-auto.py, calculates cyclomatic complexity for every function during attribute extraction. McCabe prescribed minimizing cyclomatic complexity and recommended not exceeding a value of 10 [McCabe 1976]. The role of cyclomatic complexity in predicting likelihood of defects is mixed with some studies showing a positive correlation between higher cyclomatic complexity and defect counts and other studies proving no correlation between the two. Inclusion of this metric into the possible set of modifiable variables adds the ability to study this correlation with genetic algorithms.

Test Case Execution Time (R)

Variable R represents the execution time for an application processing a negative test case. This execution time is calculated by wall clock time where the clock starts when the program under test begins execution, and it stops when the program terminates. This variable is helpful for rewarding test cases that run for longer time periods and punishing test cases that exit early. Test cases with low run times may signify error handling code catching a test case prematurely.

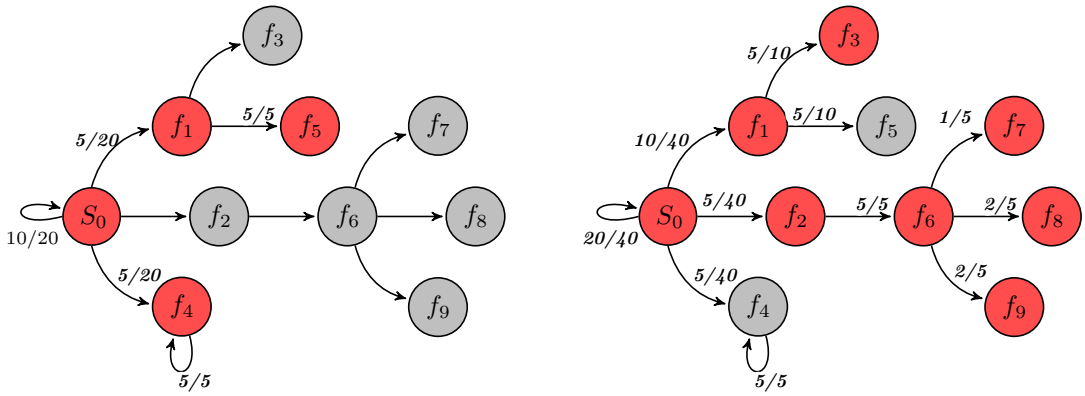
Execution Path Uniqueness (U)

Variable U represents an execution path's uniqueness as calculated from a weighted directed graph derived from a shared library's control flow graph. To explain this variable, let us first define how to create a Markov chain from a shared library. In Chapter 2, Sparks et al. [2007] demonstrated with the Sidewinder fuzzer how a control flow graph can be treated as a Markov chain. The vertices of a control flow graph represent assembly basic blocks and edges are control flow transfers (e.g. jumps and branches) between blocks. A Markov chain can be constructed by assigning transition

probabilities to control flow graph edges derived from observations of program execution. These observations record the assembly blocks executed, the number of times each block is executed, and the number of transfers between each assembly block. Sparks et al. [2007] also added two absorbing states (acceptance state and rejection state) to their graph. These states permitted the Sidewinder fuzzer to target vulnerable functions (strcpy()) by defining these vulnerable functions as acceptance states and assign any state without a path to an acceptance state as a rejection state.

A coarser control flow graph can be constructed by graphing transitions between functions instead of between assembly blocks. In this graph, vertices are functions and edges record function calls. A formalization of this coarser model, used by this dissertation, is a weighted directed graph $G = (V, E)$ where $V = \{f_1, f_2, f_3, ..., f_n\}$ and E is the set of edges connecting vertices in V. f_i in V correspond to a function in a shared library, and n is the total number of functions identified by statically analyzing a shared library. The edges between vertices are function calls within a shared library when a program processes a negative test case.

These definitions of V and E accurately model execution of functions within a shared library, yet a program calls a shared library function from code defined outside of the shared library. Shared library functions may also call functions not defined by the shared library. The current vertex set (V) does not account for these situations. To account for these undefined states, the vertex set must include a special vertex (S_0) where S_0 represents any function not implemented by a shared library. This node serves as a starting state for the directed graph (G). The amended vertex set is now $V = \{S_0\} \cup \{f_1, f_2, f_3, ..., f_n\}$. Figure 5.4 illustrates a control flow graph gathered from two distinct execution traces of the same shared library.



(a) Execution Trace One **(b)** Execution Trace Two

Figure 5.4: Function Control Flow Graph

Each weighted directed graph in Figure 5.4 has $V = \{S_0\} \cup \{f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_9\}$. Grey vertices denote functions not called while tracing execution for the current test case, and red vertices

highlight functions called during program execution with the current test case. The grey vertices and transitions to these vertices are included in both graphs to demonstrate how edge weights are modified as more negative test cases are executed. They illustrate how historical data from previous test case runs influence current edge weights and the calculation of function path uniqueness. Since historical data influences function path uniqueness, edge weights are recalculated after each negative test case.

For the control flow graph in Figure 5.4a, the observed function calls are $E = \{(S_0, S_0), (S_0, f_1), (S_0, f_4), (f_1, f_5), (f_4, f_4)\}$, and the observed function calls for Figure 5.4b are $E = \{(S_0, S_0), (S_0, f_1), (S_0, f_2), (f_1, f_3), (f_2, f_6), (f_6, f_7), (f_6, f_8), (f_6, f_9)\}$. As mentioned previously, these control flow graphs can be converted into a discrete-parameter, finite-state Markov chain by assigning probabilities to each of the edges. If each negative test case is treated as a sample of all possible execution traces, then stationary probabilities can be assigned to control flow graph edges based on the current set of samples [Sparks et al. 2007]. However, this dissertation does not define absorbing acceptance or rejection states, because the metric being calculated (function path uniqueness) does not direct execution towards any predetermined state. Also, labeling a function as an absorbing state would require adding a transition to itself in the directed graph. By adding a loop to the function, the directed graph implies a function makes a recursive call which may not be correct. The directed graphs assembled for function path uniqueness are similar to Markov chains in that probabilities are associated with edges. However, those edge weights are used to measure the uniqueness of execution paths, rather than being part of a stationary probability computation.

To assign edge weights, shared library functions are monitored as a program runs a negative test case. The monitoring reports the number of times a function was executed and the number of calls to other functions. When execution ceases processing a negative test case, weights are assigned to each edge by the formula:

$$p_{ij} = t_{ij}/v_i \tag{5.5}$$

where p_{ij} is the weight of the edge from vertex i to j, t_{ij} is the number of transitions from i to j, and v_i is the total number of transitions from state i. After each negative test case, the edge weights are updated to incorporate information gathered from past test cases and the current test case. These weights can be summarized by a matrix. Figure 5.5 summarizes the function calls collected from running two negative test cases depicted by the control flow graphs in Figure 5.4.

Notice in Figure 5.5a the matrix is sparse and populated only by function calls found in Figure 5.4a. In Figure 5.5b, the matrix combines tracing information collected from both execution traces. An example of combining information is the transition from state S_0 to state f_4. This

transition did not occur during the second execution trace, but the edge weight is updated in the control flow graph. The number of observed transitions from S_0 to f_4 in execution trace one also impacts the edge weight calculated for the observed transition from S_0 to f_1 with execution trace two.

	S_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9
S_0	10/20	5/20	0	0	5/20	0	0	0	0	0
f_1	0	0	0	0	0	5/5	0	0	0	0
f_2	0	0	0	0	0	0	0	0	0	0
f_3	0	0	0	0	0	0	0	0	0	0
f_4	0	0	0	0	5/5	0	0	0	0	0
f_5	0	0	0	0	0	0	0	0	0	0
f_6	0	0	0	0	0	0	0	0	0	0
f_7	0	0	0	0	0	0	0	0	0	0
f_8	0	0	0	0	0	0	0	0	0	0
f_9	0	0	0	0	0	0	0	0	0	0

(a) Matrix One

	S_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9
S_0	20/40	10/40	5/40	0	5/40	0	0	0	0	0
f_1	0	0	0	5/10	0	5/10	0	0	0	0
f_2	0	0	0	0	0	0	5/5	0	0	0
f_3	0	0	0	0	0	0	0	0	0	0
f_4	0	0	0	0	5/5	0	0	0	0	0
f_5	0	0	0	0	0	0	0	0	0	0
f_6	0	0	0	0	0	0	0	1/5	2/5	2/5
f_7	0	0	0	0	0	0	0	0	0	0
f_8	0	0	0	0	0	0	0	0	0	0
f_9	0	0	0	0	0	0	0	0	0	0

(b) Matrix Two

Figure 5.5: Transition Probability Matrices

Now that the control flow graph has edge weights assigned to the observed function calls, a metric can be devised to measure the uniqueness of a call trace. Sparks et al. [2007] proposed equation 5.6 as a method for calculating uniqueness.

$$f(x) = \frac{1}{\prod_{n=1}^{l} p_i} \tag{5.6}$$

In this equation, l is the length of recorded transitions for a single negative test case and p_i is the probability of each transition occurring. By taking the product of executed transition probabilities and dividing one by this product, the equation rewards a test case executing rarely taken paths [Sparks et al. 2007]. However, this equation suffers from underflow when a recorded execution path is sufficiently long. To combat underflow, this research uses equation 5.7.

$$f(x) = -\sum_{i=1}^{l} \log p_i \tag{5.7}$$

Like equation 5.6, l is the length of the recorded transitions (number of function calls recorded) and p_i is the edge weight associated with the function call paths. By summing the logarithm

of these edge weights, the equation can still reward test cases executing rarely used paths while avoiding underflow. With equation 5.7, as well as equation 5.6, higher values of $f(x)$ denote fitter chromosomes.

Function Coverage (V)

Variable V represents the percentage of functions executed while running a test case. This percentage is calculated by dividing the total number of unique functions executed by the total number of unique functions in a shared library. An increase in function coverage means a test case executed more of a shared library's functions. Executing more functions may increase the likelihood of eliciting an error. However, since a program can use a shared library but not call every function within the shared library, it may not be feasible to achieve high percentages of function coverage.

Fitness Function Implementation

Mamba implements fitness function evaluation by using an ANother Tool for Language Recognition (ANTLR) parser. The framework includes an ANTLR grammar specification defining valid mathematical operators and valid variables. The grammar permits mathematical operators +, -, *, /, ^, (), and natural log. Real numbers can be added to equations as well as modifiable and unmodifiable variables. The grammar also preserves order of operations and requires specifying a mathematical equation in infix notation.

After a program under test ceases executing a negative test case, the ANTLR parser tokenizes the configured fitness function and resolves variables through Ruby's metaprogramming facility. Variable resolution occurs by the parser querying the Ruby Plympton module's Object class to determine if a method is defined matching the current variable's name. If a method exists, then it is called with the appropriate parameters. The method then calculates the value represented by the variable by consulting extracted attribute information and hit tracing information. After the value is resolved, the method returns the calculated value back to the parser. Once received by the parser, it substitutes the value into the equation and continues evaluating the remaining expression. Once the parser finishes its evaluation, the result of the evaluated expression is returned to the genetic algorithm fuzzer as the chromosome's fitness.

5.5 Performance

The Mamba fuzzing framework now contains several genetic algorithm fuzzers as described previously. While each of these genetic algorithms perform well on certain domain specific search problems, which genetic algorithm is best suited for finding software defects? To investigate this question, each genetic algorithm fuzzer's performance must be analyzed. A common practice for vetting a fuzzer is by a direct comparison with other fuzzers that have proven capable of uncovering software defects. For this comparison, the steadfast mutational fuzzer, Mangle, and the prolific generational fuzzer, Peach, have been selected for benchmarking. Over the years, each of these fuzzers have found many defects in a plethora of programs. Comparing multiple genetic algorithm fuzzers against these time-tested fuzzers helps ascertain each genetic algorithm's aptitude for locating software defects.

To compare these fuzzers against Mamba's genetic algorithm fuzzers, a set of meaningful metrics must be defined to measure fuzzer performance. Some individuals have proposed measuring performance based on coverage of previous vulnerabilities, amount of program input space generated, or the number of interfaces supported (file, network, command line, etc.) [Takanen et al. 2008]. While these metrics are interesting from several perspectives, in my opinion, two important metrics are a fuzzer's average negative test case execution time and the number of unique defects found. Average test case execution time accurately informs a tester on the amount of test cases that can be generated and executed in a finite time period. Fuzzer designers strive to decrease average execution time, because if a fuzzer can execute more negative test cases in a shorter time frame, then the chances of it finding a software defect are likely increased.

However, one fuzzer generating and executing more negative test cases than another fuzzer does not guarantee a better rate of return. If negative test cases generated by a slower running fuzzer are more prone to uncover defects than a fuzzer generating and executing more test cases in the same time period, then the slower fuzzer's higher defect count promotes running the slower fuzzer. In the best case, a fuzzer should generate and execute as many quality negative test cases as possible in a reasonable time period. Unique defect count, when normalized on a program or set of programs across multiple fuzzers, is a reliable measure of an individual fuzzer's ability to unearth defects. Due to these reasons, all fuzzers tested by this dissertation are measured and analyzed based on both of these metrics.

Also, a testing environment for fuzzer comparison must be standardized to ensure each fuzzer runs in the same environment. The standardization guarantees each fuzzer runs on the same operating system version and tests the same version of an application. Standardization ensures a fair assessment of each fuzzer's abilities. The testing environment created for experiments

conducted by this dissertation is the Mac OS X 10.6.5 build 10H574 operating system, and all fuzzers generate negative test cases for the Preview application version 5.0.3 (504.1). The test environment includes four Mac Pro desktops of heterogeneous hardware configuration installed with these software versions. Table 5.4 documents general hardware configurations of each node in the testing environment. The environment is purposefully heterogenous, because a heterogeneous testing environment replicates practical usage of the Mamba framework and gathers metrics from multiple machine configurations.

Table 5.4: Test Environment Setup

Hostname	Processor	RAM
ARP	Quad-Core Intel Xeon 2.6 MHz	5.0 GB
GARP	Quad-Core Intel Xeon 2.6 MHz	1.0 GB
RARP	Quad-Core Intel Xeon 3.0 MHz	9.0 GB
SLARP	Quad-Core Intel Xeon 3.0 MHz	1.0 GB

For this performance evaluation, Mangle, Peach, and Mamba's four genetic algorithms along with a slightly modified genetic algorithm, SGA-MIP, conducted six different tests on each of the four fuzzing machines. On each machine, each fuzzer independently generated and executed 500, 1000, 1500, 2000, 2500, and 5000 negative test cases for the Preview application. Experiments recorded the amount of time required for each fuzzer to generate and execute each set of test cases on each machine. After recording these times, the average execution time for a single test case was calculated for each set on each machine. Figure D.1 and Figure D.2, found in Appendix D with all other graphs, plots average execution time observed for each set of test cases on the four different machines for Mangle and Peach.

These execution time graphs depict average execution time for a single negative test case to be around 4 seconds for each fuzzer. Mangle's average execution time consistently remains around 4.4 seconds on each test machine for each set of test cases. The execution time slightly increases and decreases as the number of negative test cases increases with one anomaly encountered on the GARP computer's 2000 negative test case set. For Peach, average execution time varies more than Mangle's. On each test computer, average execution time rose and fell as the number of negative test cases increased, but, overall, it still remained consistently around 4 seconds per test case. This variation between fuzzers is explained by the different attack heuristics implemented by each. Peach inserts targeted attack heuristics, known troublesome values, systematically into different sections of a PDF model. In Peach's case, average execution time depends upon the location and value of the inserted attack heuristic. Mangle's attack heuristic algorithm deviates from Peach's more precise

fuzzing by altering more bytes randomly throughout a negative test case. This coarser approach tends to execute error handling code in Preview more often which halts processing a negative test case early. These error checks smooth Mangle's average negative test case execution time.

Figures D.3, D.4, D.5, D.6, and D.7 illustrate the average execution time of a negative test case for Mamba's SGA, SGA-MM, SGA-MIP, BGA, and CHC-GA algorithms. The SGA algorithm and its variants were configured for a population size of 500, crossover rate of 30%, mutation rate of 50%, fitness function of Ba, and shared library tracing for libFontParser.dylib. A population size of 500 was chosen to introduce a significant amount of variability within the initial population. Larger populations with higher variations between chromosomes can help prevent premature convergence, because the algorithm can select diverse parents. A conservative crossover rate of 30% permits the algorithm to retain more chromosomes from the previous population without alteration. Since crossover, applied to files as chromosomes, is a destructive operation, children derived through crossover have a higher probability of being halted by error handling code. The mutation rate is set higher than the crossover rate, because mutation is a less destructive operation than crossover. Random insertion of an attack heuristic at a byte offset is not as destructive as mixing parts of two files. The only exception to this operator comparison occurs with SGA-MM. With this algorithm, the Mangle algorithm as the mutator is as destructive as crossover, but the mutation rate was left the same for consistency. The fitness function tried to maximize the average number of local variables (Ba) for all function executed in the font parsing library, libFontParser.dylib. This library is historically known for containing defects [Mitre 2010].

For BGA, all parameters remained the same except for the mutation rate. This parameter significantly decreased to 0.1% due to the change in mutation semantics between SGA and BGA. Since BGA applies the mutation rate to every gene in a chromosome, the mutation operator is highly destructive. To limit the probability of error handling code halting execution, this rate was decreased. The CHC-GA was configured to have the same effect as the BGA algorithm. Since CHC-GA does not have a mutation rate, the crossover rate was decreased from 50% (HUX) to only 0.2%. This rate only allows 0.2% of parent's different genes to be copied into a child. Once again, the lower rate tries to lessen the probability of error handling code halting processing of derived children.

As illustrated by these graphs, all genetic algorithm fuzzers add overhead to average negative test case runtime. Average execution time is increased by 6 to 10 times the average execution time of Mangle or Peach. This increase is due to these algorithms leveraging Valgrind emulation to trace shared libraries. Valgrind emulation by itself slows down execution time from 2.5 times to 93 times normal program execution time [Nethercote 2004]. The tool developed for this dissertation, Rufus, is a lightweight hit tracing tool, and, as such, its execution time performance impact aligns with

Valgrind's lesser invasive tools, Nulgrind and Addrcheck [Nethercote 2004]. It is anticipated that slowdown associated with the Rufus tool should not exceed 20 times normal program execution time.

The graphs also depict another interesting trend based on the number of negative test cases generated. As this number increases, SGA's average execution time decreases almost linearly, BGA's average execution time gradually decays towards 20 seconds per test case, and CHC-GA decreases sharply after the first generation and increases periodically as the number of generations increases. These decreases, in general, are attributable to error checks in Preview. As these genetic algorithms produce new generations, the populations of these generations slowly include corrupted files with more attack heuristics. Preview is likely to discard more corrupt files earlier in processing, because additional error checks are being encountered. This phenomenon causes the average negative test case execution time to slowly decrease as the number of test cases generated increases. For CHC-GA, periodic increases in execution time are attributable to its cataclysmic mutation operator. As the population converges, cataclysmic mutation adds variability back into the population by inserting chromosomes from the initial population. Since these chromosomes are not corrupted, Preview runs the test cases for longer time periods causing upward movement in average execution time.

At this point, based purely on average execution time, traditional fuzzers outperform genetic algorithm fuzzers. Mangle and Peach can clearly generate and execute more negative test cases for Preview in a shorter time period than any of Mamba's genetic algorithms. However, an interesting result emerges from an analysis of the number of unique defects found per fuzzing algorithm. Figure D.8 illustrates the number of unique defects found for each algorithm from all testing runs across all test machines. The graph depicts an approximate 6 to 8 times increase in the number of unique defects found between Mangle and the SGA, CHC-GA, and BGA algorithms. When compared against Peach, these genetic algorithms find between 2 and 3.5 times the number of unique defects. A comparison of all the algorithms shows Mangle finding the least number of unique defects with 2 segmentation violations. SGA-MIP followed with 2 segmentation violations and 1 floating point exception. Peach beat SGA-MIP by finding 4 segmentation violations and 1 floating point exeception. SGA-MM ranked fourth overall, by this metric, with 7 segmentation violations. CHC-GA ranked third overall with 11 segmentation violations and 1 floating point error. SGA and BGA each found 13 and 17 unique defects. Both of these algorithms uncovered multiple segmentation violations, bus errors, and floating point exceptions.

Moreover, all of the defects found by Mangle and most of the defects found by Peach are subsets of defects found by all the genetic algorithms. Peach uncovered one segmentation violation not found by any of the genetic algorithms. SGA, CHC-GA, and BGA have some overlapping defects, but the defects of SGA and CHC-GA are not a proper subset of BGA. Out of all of the unique defects found by all algorithms, five were determined to be possibly exploitable, meaning malevolent

individuals could run arbitrary code, by Apple's crashwrangler software. Mangle uncovered one of these exploitable defects, and the genetic algorithms found the rest of the exploitable defects. Multiple defects were found in the font parsing library, and the rest were found in other shared libraries used by Preview.

Based on the number of unique defects found, Mamba's genetic algorithms appear promising for finding defects in Apple's Preview application. These algorithms outperformed Mangle and Peach in finding unique defects when the number of generated test cases is limited. Moreover, Mangle and Peach's poorer performance may be due to their algorithms. Mangle alters many bytes of a test case, so most negative test cases are likely to be too corrupt for processing. Preview may catch these cases early in processing and not permit them to reach deeper sections of code. This theory is corroborated by the observed poor performance of the SGA-MIP and SGA-MM genetic algorithms, since Mangle is used to alter chromosomes in these genetic algorithms.

Peach, on the other hand, replaces data in a predetermined sequential algorithm. By limiting the number of test cases generated, Peach may not reach its more sophisticated attack heuristics. Since the genetic algorithms slowly and arbitrarily corrupt files, especially CHC-GA and BGA, over several generations, these algorithms create varied negative test cases with smaller amounts of corruption. These negative test cases are less likely to be caught by Preview's error handling code early in file processing. Once again, this explanation is supplemented by observing that the SGA-MM and SGA-MIP fuzzers uncover less unique defects than either SGA, CHC-GA, BGA.

While Mamba's genetic algorithms show promise with a small number of negative test cases causing many faults, the average negative test case execution time prohibits recommending these algorithms as feasible strategies. In order to make these algorithms attractive, the average execution time must be decreased to a level competitive with Mangle and Peach. By decreasing average execution time and combining their ability to unearth more defects in Preview with fewer test cases, Mamba's genetic algorithms may prove beneficial for fuzzing. The next chapter, Chapter 6, introduces a strategy for combating high average execution time.

Chapter 6

Distributed File Fuzzing

Genetic algorithms proved promising in the previous chapter when fuzzing Apple's Preview application. They exceeded other popular fuzzers, Mangle and Peach, at uncovering unique defects. However, a comparison of average negative test case execution time revealed genetic algorithm fuzzers incurring a 6 to 10 times increase in average execution time over other fuzzers. This incurred overhead is mainly a symptom of emulation. While these algorithms are promising in terms of unique defects found, for genetic algorithms to be competitive, they must mitigate increases in average execution time due to emulation.

One approach to combating an increase in execution time is to determine an algorithm's complexity, find and eliminate inefficiencies, or find opportunities to distribute time intensive tasks to multiple processors. To paraphrase a corollary of Amdahl's Law, make the common or most time intensive case fast [Amdahl 2000] [Tanenbaum and Goodman 1998]. Let us review Mamba's SGA algorithm to determine where improvements are possible, and if these improvements lead to a decrease in execution time.

Examining the 5,000 test case run of SGA with 500 population members reveals a total execution time for seeding an initial population, evaluating all chromosomes for fitness, and generating the next population based on their fitness to be 14,504 seconds. This measurement is sampled from the ARP computer's 5,000 test case run shown in Figure D.3. Dissecting this execution time further divulges that seeding an initial population requires 50 seconds, chromosome fitness evaluation lasts for 14,434 seconds, and reproduction takes 20 seconds. Clearly, improvements made with chromosome fitness evaluation should positively impact average negative test case execution time.

Fortunately, improvements in chromosome fitness evaluation are possible, because this component of a genetic algorithm can be distributed. Chromosome fitness evaluation is an independent task meaning each chromosome is independently tested and measured by a pre-defined fitness function.

Results from one chromosome's fitness evaluation do not alter the results of another chromosome's fitness. By distributing this component, multiple processors can independently evaluate sets of chromosomes thereby partitioning a genetic algorithm fuzzer's most computationally intensive task.

Applying this approach to Mamba's SGA example revises observed execution time by dividing it into two components, serial and parallel. SGA's serial component ran for a total of 70 seconds (0.5% of observed execution time), while SGA's parallel component absorbed most of the observed execution time with 14,434 seconds (99.5% of observed execution time). Based on these observations, Mamba's SGA algorithm should be capable of achieving improved performance, in terms of speedup, by distributing chromosome fitness evaluation, namely program emulation. A version of Amdahl's Law, shown in Equation 6.1, can help estimate anticipated execution time based on parallelization.

$$Speedup = fT + \frac{(1-f)T}{n} \tag{6.1}$$

In this equation, variable f represents the fraction of time required by an algorithm's serial component. Variable T corresponds to the observed execution time on one processor, and variable n is the number of processors assigned to process the algorithm's parallel component [Tanenbaum and Goodman 1998]. For Mamba's SGA example, f equals 0.005, T equals 14,504 seconds, and n is defined to be 4. This assignment of n means SGA's distributed version distributes chromosome fitness evaluation to four processors. Substituting these values into Equation 6.1 yields an anticipated execution time of 3,680.64 seconds for evaluating and producing one generation. Dividing this result by the total number of negative test cases executed per generation (500) produces an average execution time of 7.36 seconds.

These calculations highlight a performance bottleneck for Mamba's genetic algorithm fuzzers. By distributing chromosome fitness evaluation to only four processors, theoretically, a revised version of Mamba's SGA algorithm may realize a speedup ratio of 3.92. However, these calculations fail to account for communication overhead or inefficiencies introduced as a result of distributing algorithm components. While these calculations do not reflect possible overhead, these optimistic calculations do provide a reason to devise a distributed fuzzing architecture. They also set a goal for a distributed architecture to strive to achieve. The remaining portion of this chapter designs a distributed architecture striving to achieve the anticipated speedup ratio.

6.1 Architecture

Distributing an algorithm requires careful planning to avoid introducing bottlenecks between multiple processors or algorithm components. When distributing an algorithm, one of the main goals,

aside from avoiding bottlenecks, is to increase efficiency leading to a decrease in overall execution time. To achieve this goal, an algorithm's designer must identify an algorithm's parallel components, then devise a strategy for how multiple processors will communicate and co-operate to ensure efficient and scalable processing. Efficiency, in this instance, measures utilization of multiple processors as derived from observed speedup. This performance metric illustrates perceivable bottlenecks in communication between multiple processors and actual processor usage. Scalability, as measure by speedup, shows that the addition of more processors will decrease execution time.

As illustrated in this chapter's introduction, in the Mamba framework, distributing a genetic algorithm's chromosome fitness evaluation can potentially reduce average negative test case execution time. For this distribution, chromosomes must be divided between multiple processors for fitness evaluation, and these fitness metrics must be recorded for biasing reproduction. Since reproduction selects from an entire population based on fitness and combines chromosomes yielding a new population, a serial processing environment supports reproductive tasks better. It is conceivable to distribute reproduction to create a completely parallel genetic algorithm [Sivanandam and Deepa 2007a], but distributing chromosome fitness evaluation achieves a majority of efficiency and speedup gains. Based on these facts, Figure 6.1 illustrates a generalized distributed architecture for Mamba's genetic algorithms based on the master slave parallelization method [Sivanandam and Deepa 2007a].

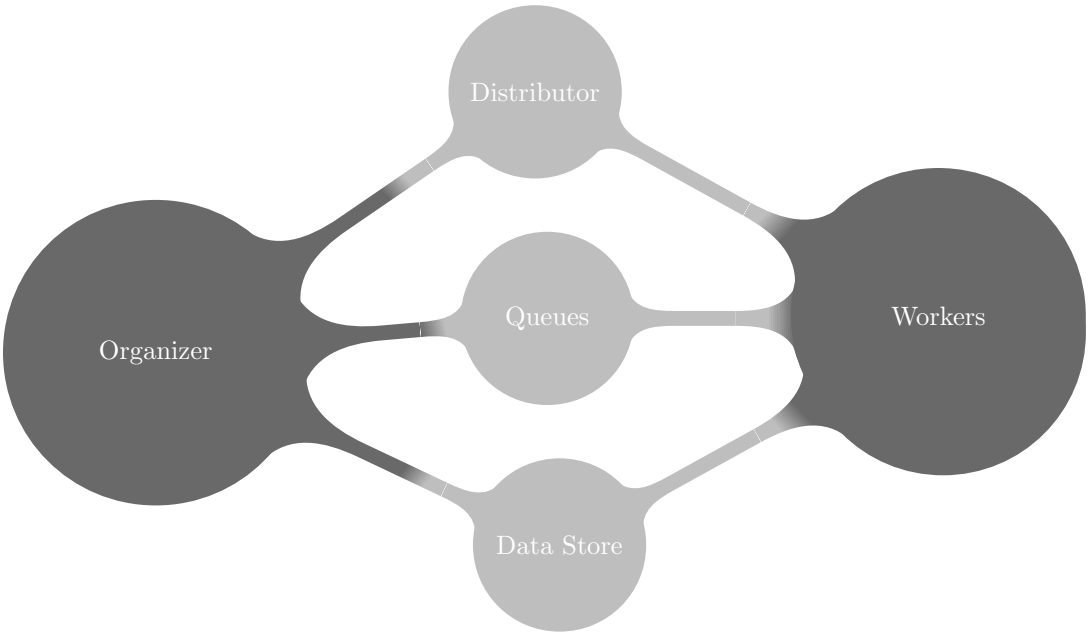


Figure 6.1: Distributed Fuzzer Architecture

As shown by the figure, Mamba's distributed architecture consists of five components. The first of which is a fuzzing organizer. An organizer is the master of a fuzzing job serving as a central command

and control node for an entire fuzzing cluster. Organizers are responsible for creating a fuzzing job, posting information about a job through a distributor, handling command and control of all nodes through managed queues, aggregating chromosome fitness metrics, performing reproduction, storing negative test cases in a data store, and halting fuzzing on all nodes when complete. Organizers are arguably the most vital component of a Mamba distributed fuzzing job.

To improve usability, Mamba implements multiple thor tasks for organizer functionality. Before using these tasks, a distributed fuzzing environment must be created by running the mamba command line utility with the -d option. This option causes Mamba to append additional parameters to the global configuration file, Mamba.yml. Figure 6.2 displays an example of a distributed configuration file.

```
---
:app: /Applications/Preview.app/Contents/MacOS/Preview
:executor: appscript
:os: darwin10.7.0
:timeout: 0
:type: DistributedCHCGeneticAlgorithm
:uuid: fz6fe26cc4ed2a11e091700026b0fffee7
:organizer: true
:server: fuzz.io
:port: 27017
```

Figure 6.2: Mamba.yml Distributed Global Configuration File

Aside from the normal parameters, the first addition to this configuration file is the organizer flag. This boolean value directs Mamba to run in either an organizer or worker operational mode. Since these modes share code sections within Mamba, the organizer flag directs Mamba to execute different code paths when necessary. In the absence of this flag, Mamba always executes as a worker. The next parameter, server, sets the Internet Protocol (IP) address or hostname for a server in which both the organizer and worker communicate. This server houses a distributed data store for negative test cases and queues for command and control. An organizer can house both of these components, and, to encourage operating in this manner, Mamba installs all required infrastructure for these components when someone installs the Mamba gem. However, Mamba provides the flexibility to run these components independently in order to increase scalability and dedicate a server specifically for data storage and queueing. Running these services on different servers aligns with Microsoft's distributed file fuzzing architecture [Conger et al. 2010]. The last added parameter, port, defines the port on the server to connect for the data storage component.

As briefly mentioned in Chapter 4, the uuid parameter serves an important role in distributed fuzzing. It defines a unique identifier for a fuzzing job, and the distributed data store and

queueing systems use the uuid in their naming conventions. By using this unique string to name the data store and queues, an organizer can host multiple instances of distributed fuzzing jobs. This unique string also composes part of the file name for Mamba package files. Package files are a convenient mechanism for distributing configurations to other nodes within a fuzzing cluster. An organizer generates these files after their environment is setup. Configuring an organizer consists of changing algorithm specific configuration file parameters, storing shared library disassemblies in the environment's disassemblies directory, and creating an initial population seed file in the environment's tests directory. Once the environment is configured accordingly, running the '`thor fuzz:package`' task creates a package file. This task generates a zip file named uuid.mamba where the uuid is the unique identifier defined in the Mamba.yml configuration file. To create this file, the task compresses all files contained in the fuzzing environment's configs, disassemblies, and tests directories. The task also alters the organizer parameter in Mamba.yml to be false, since an organizer distributes this package file to workers only. After package file creation, an organizer uses the next distributed model component, a distributor, to dispense the package file to potential workers.

A distributor is an entity or method for relaying packaged configurations to interested workers. Distributors are meant to tackle a problem encountered by many independent security researchers. This problem is a requirement for a distributed homogenous environment for generating and running negative test cases against an application. While this requirement is not necessarily required for a centralized fuzzer, in a distributed environment, every node's configuration should be consistent to guarantee reproducibility of crashes. Several hurdles may impact the ability for a security researcher to create a homogenous environment. These issues include financial barriers due to multiple hardware instances or licensing issues associated with running multiple instances of an operating system. While virtualization has decreased costs, setup and maintenance of a distributed fuzzing environment remains a factor.

A method for alleviating these issues is to provide an open call for other individuals to join in testing thereby distributing associated costs, licensing, and maintenance across multiple users. This method is similar to popular distributed computation platforms like SETI@home. The user collective, crowd, can then build a private fuzzing network by downloading or trading Mamba packaged configurations. A distributor is the method or centralized distribution point for providing an open call for participation. Example distributors can be as high tech as a web application where willing participants signup to contribute CPU time [Conger et al. 2010] or as low tech as emailing Mamba package files to a closed group. For this research, distribution was achieved by copying packaged Mamba configurations to a controlled worker group. However, Mamba's distributed model does permit other distributor types without any alteration, and this area is open for further research.

Once a worker discovers a fuzzing job worthy of joining and their environment matches the organizer's environment, an individual controlling a worker node downloads the Mamba package file. A matching environment means the worker node runs the same operating system version and application version being tested by the organizer. After the package file is downloaded, the worker creates a fuzzing environment with the mamba command line utility and copies the package file into the environment. During creation, the fuzzing environment's type (-y option) is inconsequential, because Mamba implements a thor task, '`thor fuzz:unpackage`', that erases the current environment's configuration, unzips the packaged file, and configures the environment to work with the organizer. All files packaged by the organizer, configs, disassemblies, and tests, are installed on the worker by using the unpackage thor task. Once unpackaged, the worker can then contribute to the distributed fuzzing environment by running, '`thor fuzz:start`'. Packaging and distributing Mamba configuration information lowers setup time and potentially opens co-operative distributed fuzzing to a larger audience. It can also reduce software dependencies, such as IDA Pro, since a fuzzing organizer already packages and exports disassembly information with Mamba package files.

Organizers, workers, and distributors are conceptually complete at this point, but key components of a distributed system are still missing. Distributed systems must have the ability to divide work between multiple processors and share data. In Mamba's distributed architecture, division of labor and control mechanisms originate from the organizer, and the organizer distributes these messages via a queueing system. A centralized data store accumulates shared data to relay between an organizer and workers. For these components, Mamba installs and uses Rabbitmq and Mongodb, respectively.

Rabbitmq is an implementation of the Advanced Message Queuing Protocol (AMQP) specification. AMQP defines an open standard for peer-to-peer messaging. The standard prescribes an encoding for peer-to-peer messages and a layered messaging architecture built upon the message encoding. This layered architecture adds transport mechanisms, formatting, transactions, and authentication and encryption mechanisms [Group 2011]. Rabbitmq implements this standard with the functional programming language, Erlang.

Before delving into Mamba's Rabbitmq usage, a few terms specific to AMQP must be defined. AMQP's messaging architecture behaves much like a producer and consumer messaging model. Producers generate messages in AMQP format and deliver those messages to an exchange. An exchange is akin to a mailbox or namespace with semantics associated to an exchange based on its type. Mamba utilizes two of the three possible exchange types, direct and topic. A direct exchange is analogous to unicast routing where a producer sends a message to only one consumer. On the other hand, a topic exchange routes messages sent by a producer to multiple queues based on a

routing key. A routing key defines a textual pattern with or without wildcards (*) to compare with an incoming AMQP message's header. Based on the matching keys, an exchange appends a message to one or more queues assigned to the key. Queues are entities storing messages from a producer, and they deliver those messages to subscribed consumers. Queues bind to different exchanges to create routing scenarios for published messages. Consumers subscribe to these queues to receive messages [Group 2011].

For Mamba, an organizer creates a direct exchange and binds a queue named testcases to this exchange. Then, the organizer and all workers subscribe to this queue. Once a population of chromosomes, negative test cases, are ready for fitness evaluation, the organizer publishes messages to the direct exchange commanding all consumers subscribed as to which chromosomes to evaluate. Each message published to the exchange is a string of the form "generationNumber.chromosomeID". The generationNumber specifies the generation currently being evaluated, and the chromosomeID specifies a chromosome's unique identifier within the current population. Chromosome IDs are a numbering from zero to the population size minus one of all chromosomes.

Once the organizer publishes a message to the direct exchange, all consumers subscribed to the testcases queue receive these messages by round-robin scheduling. This ordering means the first subscribed consumer receives the first message, the next consumer receives the next, and so on until the queue is empty. When a worker finishes evaluating a chromosome, it acknowledges receipt and processing of it, and the worker requests another test case from the queue to evaluate. This round-robin scheduling with acknowledgements guarantees persistence of test case delegations and on-demand servicing.

Additionally, a Mamba organizer creates a topic exchange and binds four queues to the exchange. The organizer assigns each of the routing keys, commands, crashes, remoteLogging, and results, to one of the four queues. If an AMQP message is sent to this exchange with one of these keys, then the exchange delivers the message to the appropriately bound queue. All consumers, the organizer and workers, receive messages published to these queues. The command queue handles messages for halting all nodes. Once a fuzzing algorithm approaches a stopping condition, the organizer publishes a shutdown message to the exchange with the commands routing key. All workers receive this message and halt their running instances. The crashes queue notifies all participants of a negative test case causing a crash. When an organizer or worker detects a crash from a negative test case, the participant publishes a message to the topic exchange with the crashes routing key. Distributed crash notification helps crash analysis after fuzzing ceases.

The remoteLogging queue is for informational messages to be sent to all participants. These messages show up in the organizer and worker's logs after receipt. The results queue is a special queue. Once a worker calculates a chromosome's fitness, the worker posts this fitness value to

the topic exchange with the results routing key. The format of the message is of the form "generationNumber.chromosomeID:fitness". The organizer then collects these results and parses the textual string to assign fitness metrics to a chromosome. Once all chromosomes in a population are evaluated, the organizer uses these fitness measures to bias selection and produce the next generation.

The queueing system behind Mamba's distributed environment appears complex, but Mamba hides all of these intricacies from its users. Mamba implements thor tasks to start and stop the Rabbitmq component. These tasks are '`thor distrib:qstart`' and '`thor distrib:qstop`'. The start task forks a detached instance of Rabbitmq and directs it to store queuing information in the fuzzing environment's queues directory. The start task also redirects Rabbitmq's logging information to the fuzzing environment's logs directory. The stop task wraps the rabbitmqctl utility to stop the currently running instance of Rabbitmq. As will be seen with all distributed components, Mamba implements thor tasks for distributed components in the distrib namespace. Hence, a task controlling any component of the distributed environment will be of the form distrib:action.

Rabbitmq suffices for the queueing component of Mamba's distributed architecture, but it does not solve Mamba's data storage problem. Queues are good for storing short messages and delivering those messages to one or more consumers. However, queues are not good for persistent storage of data several megabytes in size. For Mamba's genetic algorithms, chromosomes are files, so chromosomes are often several megabytes. Also, if a chromosome causes an application to crash, then the chromosome must be retrieved from storage for analysis after fuzzing ceases. Mamba's distributed architecture also requires minimal configuration such that configuring a fuzzing job requires minimal system administration knowledge. Making this a requirement of Mamba lowers the threshold for any individual to host or join a distributed fuzzing environment.

Multiple software packages implement functionality required for a distributed data store. The predominant solution for distributed data storage and processing is Hadoop Distributed File System (HDFS). Unfortunately, HDFS employs a complicated configuration scheme leveraging XML files for setting configuration parameters. While this configuration scheme is appropriate for an enterprise level distributed filesystem, it is unsuitable for an average user wanting to quickly host a distributed fuzzing environment. Fortunately, the recent NoSQL movement produced a viable solution for a distributed data store with an option for minimal user configuration via the command line. This solution is the document-oriented database, Mongodb.

Mongodb is an upcoming competitor to traditional relational databases like MySQL. Released in February 2009, Mongodb provides a cross-platform solution for data storage in a format known as Binary JSON (BSON). BSON is a flexible data encoding with several basic types (byte, int, double) along with the ability to store complex data types like strings, arrays, and arbitrary

binary data [10gen 2009]. Mongodb stores BSON encoded data, in its vernacular, as documents in collections within a database. Collections are analogous to a relational database's tables, and like entries in a table, Mongodb can index, update, and query stored documents [Chodorow and Dirolf 2010].

For Mamba, Mongodb's attractive property is its GridFS specification. GridFS is a standard for storing arbitrarily large BSON objects within Mongodb. This specification assigns two collections, files and chunks, for storing and accessing these large objects. The documents within the files collection stores metadata about an object such as its filename, content type, MD5 hash, and length. Mongodb fragments large objects into smaller chunks of 256 kilobytes and stores those fragments as documents within the chunks collection. Mongodb then exposes an API for accessing a large BSON object as a file, and it handles reassembling the object's chunks stored in the chunks collection. The API hides all fragmentation, storage, and reassembly from the end user by providing functions equivalent to standard file input/output APIs. Storage and retrieval of a file then only relies on connecting to a Mongodb database and opening a file by its unique identifier or filename [Chodorow and Dirolf 2010].

Due to Mongodb's GridFS file abstraction for arbitrarily large BSON encoded objects combined with Mongodb's configuration via the command line, Mamba leverages it for a distributed data store. When an organizer begins a fuzzing job, it starts an instance of Mongodb with the thor task, '`thor distrib:dstart`'. This task forks an instance of Mongodb, directs Mongodb's logging to the fuzzing environment's logs directory, and Mongodb's data storage files to the databases directory. After the data store and fuzzing job are started, the organizer creates a database with the unique identifier found in the Mamba.yml configuration file as the database's name. The organizer then seeds the database with a genetic algorithm's initial population (i.e. chromosomes represented as files). Each document created from a chromosome sets a unique ID to be of the form "generationNumber.chromosomeID". This unique ID supports Mongodb's indexing and searching, and its format matches the format of messages sent by an organizer to the direct exchange for distributing chromosome fitness evaluation. Since these identifiers are equivalent, a consumer receiving an AMQP message from the testcases queue can then use it to query the Mongodb database for the corresponding chromosome. Once the consumer obtains a GridFS file handler corresponding to the chromosome, the consumer downloads the document and executes the application under test with the downloaded negative test case. The consumer then monitors the application's execution and records hit tracing information in a local XML file. The consumer computes the chromosome's fitness by parsing this XML file and evaluating the configured fitness function. Chromosome fitness is then uploaded to the topic exchange with the results routing key for the organizer to receive, and the consumer uploads the XML trace file to the Monogdb database. The unique identifier for

the trace file is set to "generationNumber.chromosomeID.trace". Workers then remove negative test cases and XML trace files from their local storage to preserve disk space. Uploading the trace files permits the organizer to analyze and review crashes after fuzzing finishes.

For convenience, Mamba includes two thor tasks to startup and shutdown all distributed components on an organizer. The '`thor distrib:start`' task forks a Mongodb instance and starts Rabbitmq. The '`thor distrib:stop`' task finds the currently running instance of Mongodb, terminates the process, and uses rabbitmqctl to stop the Rabbitmq queuing system. Also, during installation of the Mamba gem, the packaged gem downloads and installs Rabbitmq and Mongodb locally in the system's Ruby gems directory. By providing convenience tasks and automatically downloading and installing dependencies, Mamba alleviates some of the difficulty involved with installing a distributed fuzzing environment.

6.2 Results

The creation of a master-slave distributed architecture is an attempt to decrease average execution time for a negative test case. The aforementioned architecture creates a master (organizer) and a set of slaves (workers). To test the ability of a master-slave parallelization scheme with Mamba's genetic algorithms, a new set of experiments were conducted. These experiments replicated the initial experiments devised for testing genetic algorithm fuzzers as described in Chapter 5.

These new experiments distributed chromosome fitness evaluation among four computers within a local area network testbed (ARP, GARP, RARP, SLARP). In this configuration, ARP acted as the master for all experiments, and the other computers served as the slaves. This configuration matches the speedup calculations presented at the beginning of this chapter where n is set to four. The experiments tested the Preview application with the same parameters for each genetic algorithm as tested in Chapter 5. Each algorithm ran a set of tests generating 500, 1000, 1500, 2000, 2500, and 5000 negative test cases. The average execution time for each algorithm for each set is shown in Figure D.9. The figure also graphs serial execution time of Mangle and Peach to show that distributing Mamba's evolutionary fuzzing is required to make it competitive with other fuzzers.

In the figure, lighter colored lines plot the average execution time per negative test case observed for genetic algorithms in serial mode. Darker lines graph the distributed configuration for each algorithm as well as execution times observed for Mangle and Peach. As shown, distributed genetic algorithms decreased average execution time in most cases by at least 20 seconds when distributed amongst four computers. Average execution time for all genetic algorithms in a distributed environment measured within 3 seconds of the 7.36 second average execution time projected by Equation 6.1. Since the population size for each genetic algorithm is set for 500 chromosomes,

the graph depicts the highest execution time for each distributed genetic algorithm occurring with the initial population. This behavior is expected, because the initial population contains a limited number of corrupted files. As the genetic algorithms evolve new generations, the average execution time declines, because newer generations encompass more corrupted files. Preview's error handling code detects these corrupted files and halts execution early. As more corrupted files enter the population, the smaller execution time for the corrupted chromosomes decreases the average execution time.

When each algorithm generates the largest number of negative test cases (5000), average execution time of a negative test case is between 5.97 seconds for CHC-GA and 7.36 seconds for SGA. These measures show that the CHC-GA exceeds speedup projections of a distributed SGA. This behavior is justifiable because initially CHC-GA achieved a lower average execution time over SGA when operating serially. Overall, distributing Mamba's genetic algorithms in a master-slave parallelization scheme to four computers decreases average negative test case execution time to levels competitive with serial Mangle and Peach. When the genetic algorithms generate 5000 test cases SGA is within 2.87 seconds of Mangle and 3.72 seconds of Peach testing 5000 test cases. BGA's average execution time is within 1.65 seconds of Mangle and 2.5 seconds of Peach. CHC-GA averages execution time per negative test case within 1.48 seconds of Mangle's and 2.33 seconds of Peach's average negative test case execution time. These observed speedups increase the number of negative test cases that can be tested by a genetic algorithm fuzzer within a fixed time period, and the speedup makes the algorithms attractive options when comparing against Mangle and Peach. Distributing these algorithms to additional computers should decrease average execution time further.

Chapter 7

Fitness Function Case Study

Genetic algorithm fuzzers proved promising in Chapter 5 in comparison to Mangle and Peach. Mamba's genetic algorithms managed to uncover more defects than Mangle or Peach in the Preview application when the number of negative test cases is constrained. For the best performing genetic algorithm, BGA, the rate of defect discovery equated to 17 unique defects from 12,500 negative test cases generated. While this rate of return may be considered low, it is encouraging that these defects were found with a randomly selected initial population, and the genetic algorithm traced 1 out of 127 possible shared libraries.

In Chapter 6, distributed file fuzzing corrected an inherent flaw in Mamba's architecture. Gathering a hit trace from an application through dynamic binary instrumentation is an expensive operation. The DBI framework must translate an application from its original binary representation to an intermediate representation, instrument the IR, and compile the newly instrumented IR back into machine code. Overhead associated with these operations increase execution time significantly. While other methods may exist to gather this information, fortunately, genetic algorithms can be parallelized in a master-slave architecture. Distributing chromosome fitness evaluation, emulation, counteracted emulation's execution time increase thereby improving the outlook of genetic algorithm fuzzers using emulation.

Since Mamba's architecture is now defined with a set of genetic algorithms, the next step is to evaluate the effectiveness of multiple fitness functions. This case study of fitness functions should provide insight into the usage of Mamba's variable set when measuring negative test case fitness. Table 7.1 summarizes the five fitness functions selected for this case study.

To measure effectiveness, multiple experiments were conducted with these fitness functions and varying population sizes. The experiments catalogued the number of unique defects found by each fitness function and population size pairing. Execution time is also factored into the comparison

of fitness functions. However, it is assumed that any variations in execution time can be combated by adding more or less workers to a distributed environment using these fitness functions. Before presenting the results from this case study, let us first examine the composition and reasoning behind the fitness functions selected.

Table 7.1: Fitness Function Case Study

ID	Fitness Function	Semantics
1	Ba * R	Average Number of Local Variables Rewarding Longer Execution Time
2	Ga	Average Cyclomatic Complexity
3	Ea * R	Average Number of Instructions Per Function Rewarding Longer Execution Time
4	Ba * V	Average Number of Local Variables Rewarding Function Coverage
5	Ba / V	Average Number of Local Variables Penalizing Higher Function Coverage

The first selected fitness function is $Ba * R$. Deconstructing this expression and the modifiable variable, Ba, provides insight into the logic behind maximizing this function. The B variable represents the number of local variables for an executed function. By appending the a modifier to the variable, the variable's meaning changes to the average number of local variables for all functions executed. As shown from previous experiments in Chapter 5, multiple genetic algorithms maximizing the function Ba demonstrated an aptitude at evolving negative test cases for Preview resulting in the application crashing. Across all genetic algorithms tested in Chapter 5, maximizing this variable caused a total of 29 crashes. These crashes are especially significant, because all experiments limited genetic algorithms to generating less than 5,000 negative test cases per run equating to at most 10 generations of population size 500. A limitation on the number of negative test cases decreases the probability of randomly creating a negative test that causes an application to crash.

While maximizing this variable proved promising, initial experiments highlighted one dilemma. As genetic algorithms progressed through generations, the number of successful negative cases declined. Later generations contained a higher proportion of severely corrupted chromosomes than the initial starting population. Naturally, this phenomenon is a byproduct of reproductive operators, but these severely corrupted chromosomes provide minimal value to the overall population. Severely corrupted chromosomes, in most instances, are incapable of exercising code other than error handling routines. These chromosomes can be beneficial if an application's code base is immature, because error handling routines may not exist to gracefully handle error conditions. Adding the R variable (negative test case execution time) balances the composition of later generations by rewarding test cases with longer execution times. The equation's goal is to evolve negative test cases exercising

functions with higher averages of local variables and discourage the genetic algorithm from retaining severely corrupted chromosomes.

The second fitness function tested by this case study is Ga. The G variable represents cyclomatic complexity which is a software complexity measure [McCabe 1976]. Cyclomatic complexity, applied to an application, is based on a decomposition of code into a set of vertices and edges. The vertices signify a basic block of a program, and the edges define control flow paths between a program's basic blocks. For Mamba's application of cyclomatic complexity, a basic block is a contiguous set of assembly instructions with one entry point and one exit point. The exit point is an assembly instruction altering control flow such as a conditional or unconditional jump. This research applies cyclomatic complexity to a function instead of an entire program, so cyclomatic complexity only measures the different possible control flow paths of an individual function.

Appending the a modifier to the G variable alters the variables semantics to be the average cyclomatic complexity of executed functions. In [McCabe 1976], McCabe prescribed minimizing cyclomatic complexity to simplify code structure which could decrease or eliminate software defects. The fitness function, Ga, tests cyclomatic complexity's relevance when evolving negative test cases. Genetic algorithms configured with this fitness function will try to evolve negative tests exercising complex functions containing numerous conditionals or loops. The hypothesis being tested is, are more defects discovered when executing functions with higher complexity?

The third fitness function is based on the complexity measure E, which is the number of assembly instructions per function. This fitness function appends the a modifier to the E variable to measure the average number of assembly instructions of executed functions. By multiplying Ea by a negative test case's execution time (R), the fitness function rewards a negative test case for longer execution time and increases emphasis on a higher average number of assembly instructions per function.

The fourth fitness function, $Ba * V$, is a variation on rewarding a higher average number of local variables. Rewarding a higher average number of local variables is attractive, because programmers rely on local variables to store temporary values. Since these variables are local in scope, programmers tend to be careless with allocation and management of these variables. Many defects, experienced first hand by this author, are a direct result of local variable mismanagement. By multiplying this variable by V, function coverage, the fitness function rewards a negative test case which executes a larger percentage of functions in the traced shared library. Rewarding based on increased function coverage should cause the genetic algorithm to evolve negative test cases executing different and more shared library functions.

The last fitness function, Ba/V, checks the genetic algorithm implementation to ensure test cases are evolving as expected and results are not due to random chance. As before, maximizing Ba promotes executing complex functions using multiple local variables. However, dividing Ba by

function coverage (V) punishes negative test cases for executing a larger percentage of functions within a traced shared library. A higher percentage of functions executed causes the influence of Ba to be decreased. This function should reward corrupted chromosomes and find less defects than all other fitness functions.

While choosing fitness functions for testing is important, another consideration in the design of this case study is equally important. This consideration is the genetic algorithm to use with these fitness functions. From Figure D.8, a natural choice is BGA since it uncovered the most defects, 17, and it found defects in multiple categories (SIGSEGV, SIGBUS, SIGFPE). However, the results from using this choice are likely to be more of the same, and it is of greater interest to explore two additional factors. These factors are the genetic algorithm's population size and the number of generations to evolve. Each of these parameters can impact an evolutionary algorithm in unintended ways.

For these experiments, two population sizes are evaluated. These population sizes are 250 and 100 chromosomes. As a side note, the initial population for both of these sizes are independently acquired through Mamba's '`thor fuzz:seed`' tool. All experiments configured for a population size, 250 or 100, reuse the acquired initial population corresponding to its size. The important aspect of these population sizes are that each size is at most half the size of the populations tested in Chapter 5. Decreasing the size limits the mating pool which hinders reproduction for some genetic algorithms, and the degree to which reproduction is disadvantaged is being tested.

The other parameter, the number of generations to evolve, is set for creating 25,000 negative test cases. For populations of 100, this stopping condition means 250 generations, and for populations of 250, evolution stops after 100 generations. Comparing these conditions with the configuration of tests executed in Chapter 5 show that case study experiments will generate 5 times as many test cases. This increase questions whether some genetic algorithm variations will converge prematurely causing later generations to be useless.

Consistent with the changes in population size and the number of generations produced, this case study chooses Mamba's CHC-GA for testing. From Figure D.8, CHC-GA discovered the third most defects, 12, which narrowly fails to overtake the second place algorithm with 13 defects. Choosing CHC-GA for longer experiments is interesting, because CHC-GA implements a crossover operator similar to BGA and a cataclysmic mutation operator. The mutation operator rectifies premature convergence. Once the algorithm detects convergence, it reinvigorates evolution by copying chromosomes from the initial population. By using CHC-GA, longer fuzzing runs have a mechanism against premature convergence, and it is interesting to see how well that mechanism will perform.

7.1 Results

The test environment for this case study remains consistent with experiments conducted during development of Mamba's genetic algorithms and distributed environment. The study evaluates Preview version 5.0.3 (504.1) on Mac OS X 10.6.5 build 10H574. Since Mamba's distributed environment decreases execution time, this study uses it to distribute chromosome fitness evaluation, and all machines listed in Table 5.4 are used for distributed testing. Distribution results in experiments running between 1.44 days to 2.62 days equating to between 4.63 to 9.07 seconds per negative test case.

Execution time variation is a side effect of rewarding negative test cases for longer execution times. For instance, the fitness function, $Ea * R$, with a population size of 100 recorded the longest average negative test case execution time. The smallest average negative test case execution time belonged to the fitness function, Ba/V, since this function rewarded chromosomes executing a smaller percentage of shared library functions. No trends in average execution time emerged from testing the same fitness function with different population sizes. In the case of the fitness function, Ga, average execution time for a population size of 250 was 5.91 seconds and 4.98 seconds for a population size of 100. Fitness function, $Ea * R$, recorded average execution time for a population of 250 chromosomes as 7.11 seconds. This recorded average is smaller than the average execution time for a population of 100 chromosomes with the same function (9.07 seconds).

As for unique defects discovered, Figure D.10 and Figure D.11 illustrate the number of unique defects found by each fitness function for population sizes of 250 and 100, respectively. All CHC-GAs with a population size of 250 found more defects overall than their 100 population size counterparts. Additionally, the fitness function, $Ba * R$, performed well in both configurations. With a population size of 250, it found six segmentation violations, and with a population size of 100, the algorithm discovered four segmentation violations.

The fitness functions, $Ea * R$ and $Ba * V$, managed to discover six defects each when configured with a population size of 250. The first function found five segmentation violations and one floating point exeception, while the second function uncovered six segmentation violations. Performing the tests again with a population size of 100 caused the $Ea * R$ function to find one segmentation violation and the $Ba * V$ function to uncover two segmentation violations. These results imply that rewarding a higher average of local variables may be capable of finding more defects than other defined variables regardless of population size. This is supported by the $Ba * R$ function finding four segmentation violations when configured for a population size of 100. The results also highlight a trend with each CHC-GA configured with a smaller population size. For all case study experiments,

a smaller population size caused each genetic algorithm to find less defects due to minimal diversity in the mating pool.

The Ga fitness function performed almost as well as $Ea*R$ and $Ba*V$ by finding five segmentation violations when configured with a population size of 250. This function followed the same trend in finding less defects in its smaller population size configuration. In that configuration, the function found only one segmentation violation. While this function did not outperform $Ea * R$, the Ga fitness function produced results almost identical to $Ea * R$.

As expected, the worst overall fitness function was Ba/V. The function discovered only one defect in all experiments. This function underperforms since it punishes test cases with higher function coverage thereby reducing higher averages of local variables. These results provide evidence that maximizing different equations composed of predefined variables can influence negative test case evolution and defect discovery.

While selecting an algorithm for these experiments, the decision considered the effects of premature convergence. For experiments configured with a population size of 250, these algorithms did not enter cataclysmic mutation. Not entering cataclysmic mutation means the population size and the initial population composition was sufficient and diverse enough to prevent convergence for 100 generations. However, the algorithms configured with a population size of 100 chromosomes did require cataclysmic mutation for three of the five fitness functions. The fitness function, $Ba * R$ caused the genetic algorithm to enter cataclysmic mutation at generation 127. The fitness function, $Ea*R$, converged the population at generation 74, and the function $Ba*V$ converged at generations 74, 101, and 135. This result supports a recommendation of CHC-GA for larger fuzzing runs when the population size is sufficiently small. For this case study, sufficiently small is a population size of 100 chromosomes.

Overall, this case study's recommendation is to use the average number of local variables to direct evolution. This variable found multiple defects during genetic algorithm development and in multiple population size configurations. Code complexity measures, cyclomatic complexity and the number of assembly instructions, also proved to be valuable options. The complexity measures both uncovered almost as many defects as rewarding a higher average number of local variables. Rewarding negative test cases with larger execution times did not help premature convergence as originally hypothesized, but rewarding longer execution times did outperform rewarding higher function coverage in smaller population sizes.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

Genetic algorithm fuzzers are a viable option for generating negative test cases to exercise an application's file parsing routines. From the experiments conducted by this research, genetic algorithms discovered a total of thirty-one unique defects in Apple's Preview application. Of these thirty-one defects, genetic algorithms found four defects in the traced library, libFontParser, and four defects were found in libraries directly called from functions in the libFontParser shared library. That means 27% of all defects found during this research were directly or indirectly related to a chosen shared library. The remaining defects were found in three other shared libraries not directly related to the libFontParser library. These numbers reflect how tracing a chosen shared library and evolving negative test cases based on metrics from tracing that library can cause a genetic algorithm to target defect finding even with random attack heuristics.

Aside from discovering defects, this dissertation also contributed to the advancement of feedback fuzzing in multiple areas. These contributions include:

- *Development of a genetic algorithm fuzzing framework with multiple genetic algorithms.* - The only available genetic algorithm fuzzing framework known by this author is EFS by Jared DeMott [DeMott et al. 2007]. EFS relies on the PaiMei reverse engineering framework, and specifically, it uses pydbg from PaiMei to collect hit tracing information. PaiMei is largely written to support Windows and patches for Mac OS X do not work with current versions of Mac OS X (10.6 and 10.7). EFS also implements only one evolutionary algorithm. Mamba extends genetic algorithm fuzz testing on Mac OS X and provides multiple genetic algorithms with configurable parameters.

- *Definition of a set of variables for fitness function composition.* - This research created a set of variables to include in mathematical fitness function's for evolving chromosomes. The variables combine information from static analysis as well as dynamic analysis. By combining these domains and exposing a set of variables to a tester, the tester can now define a fitness function to suite their own requirements. This architecture also allows experimentation with different fitness functions and different genetic algorithms.

- *Proof of concept implementation of an ad hoc open source distributed file fuzzing framework.* - Distributed file fuzzing frameworks are not widely deployed today. Most distributed fuzzing frameworks are proprietary implementations by corporations or security researchers. The most widely known open source distributed fuzzing framework is Peach. The development of Mamba demonstrates how to build a distributed file fuzzing framework built upon open source components.

8.2 Future Work

This dissertation focused solely on creating a genetic algorithm fuzzing framework for Mac OS X applications. The decision to focus on applications running on Mac OS X was based on the limited support of current fuzzing frameworks for this operating system. Designing a new framework for Mac OS X presented a challenge and limited some techniques. Many tools, such as other dynamic binary instrumentation tools, Pin and DynamoRIO, or common debugger functionality, gdb watchpoints on library loads, are not currently supported or implemented on Mac OS X. These limitations caused several design decisions to be made, and other platforms may implement functionality or support other tools which would make Mamba more efficient. However, design decisions made to support Mac OS X should minimally impact porting Mamba to other operating systems.

Irregardless of any implementation specific issues, this fuzzing framework opens several areas for future research. Some of these research areas include:

- *Improve Genetic Algorithm Hit Tracing Technique* - Mamba addresses hit tracing by using Valgrind to emulate an application, and this emulation increases an application's execution time. Future research should explore other options for collecting hit tracing information, especially for Mac OS X. A likely candidate is gdb breakpoints combined with gdb's Python integration. Mamba could install a gdb catchpoint for shared library loading (currently not supported on Mac OS X), and once triggered, set breakpoints for all addresses exported from IDA Pro offset by the library's load address. Gdb's Python integration could then record XML trace information in Mamba's defined format. Other candidate tracing solutions are

Ruby's Ragweed library, a custom ptrace() library, Apple's dtrace implementation, or scripting lldb with Python. Each candidate mechanism will have design trade-offs and cross-platform concerns, but they may improve execution time and permit tracing more than one shared library.

- *Comparative Study of Genetic Operators and Genetic Algorithms* - Mamba's configuration paradigm opens the possibility of testing different rates for genetic operators. The two prominent parameters to test are crossover rate (P_C) and mutation rate (P_M). Different values for these rates may improve or diminish a genetic algorithm's effectiveness in uncovering software defects. Experiments of different rates might test all genetic algorithms within Mamba to measure effects of these parameter values on the different algorithms. Further studies could also compare Mamba's genetic algorithms by testing multiple applications for more generations. The distributed framework makes running all genetic algorithms to create over a million negative test cases feasible. Trends may emerge depicting one genetic algorithm as a better choice for fuzz testing.

- *Introduce Data Modeling* - Generational fuzzing started with Dave Aitel's seminal paper on block-based modeling [Aitel 2002a]. By modeling a file or protocol, a fuzzer can insert an attack heuristic within data based on a model. The modeling guides a fuzzer to choose a semantically correct attack heuristic. Modeling could improve genetic algorithm performance, and it could advise a genetic algorithm on crossover points. By consulting a model for crossover point selection, a genetic algorithm could forego splitting a negative test case across a model field. Model consultation also opens the ability to apply genetic programming concepts to model crossover and mutation. Representing a model as a tree would permit operations like subtree crossover and even adding a rudimentary repair operator to fix severely damaged chromosomes. Developing genetic programming concepts in Mamba could also permit testing of programming language implementation, e.g. Javascript, within web browsers.

- *Develop a Multiple-Population Parallel Genetic Algorithm* - Mamba's distributed model does not restrict usage of any of its components. It also does not constrain all worker nodes to measure a chromosome by the same fitness function. A multiple-population parallel genetic algorithm consists of multiple populations running on different processors possibly being evaluated by different fitness functions. At different intervals, these populations exchange individuals to add variation back into their populations [Sivanandam and Deepa 2007a]. Altering Mamba's distributed architecture to designate sets of chromosomes to preset processors could help in achieving a multiple-population parallel genetic algorithm. However,

distributing reproductive tasks to worker nodes is required for cultivating subpopulations along with the development of a migration operator.

- *Distributor Component Applied Research and Peer-To-Peer Fuzzing* - This dissertation did not explore the possible multiple distributor types. The most promising type, hinted by this dissertation and used by Microsoft's distributed file fuzzer [Conger et al. 2010], is a web application. A web application could be designed with a restful API to support Mamba's organizers and workers. Coalescing fuzzing jobs and advertising for help aligns with Microsoft's model as well [Conger et al. 2010], but it expands their efforts by permitting endpoints to communicate directly without an intermediary controlling all endpoints and all jobs. A study of distributors could also examine social issues with fuzzing darknets and responsible disclosure in situations where multiple entities may have collectively discovered a software defect. Mamba's distributed environment could also be improved by eliminating organizer and worker classifications. The fuzzing environment could allow any node to distribute work and provide failover when a command node stops responding. This failover would require all nodes to run Mongodb as a replica set and cluster with Rabbitmq. If each node runs these services and all nodes announce joining and leaving a fuzzing job, then Mamba's architecture could be transformed into a peer-to-peer fuzzing network.

Bibliography

Bibliography

10gen, I. (2009). *BSON Specification Version 1.0*. 10gen,Inc. 88

Aitel, D. (2002a). The advantages of block-based protocol analysis for security testing. 16, 17, 99

Aitel, D. (2002b). An introduction to spike, the fuzzer creation kit. 15

Amdahl, G. M. (2000). Readings in computer architecture. chapter Validity of the single processor approach to achieving large scale computing capabilities, pages 79–81. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. 80

Amini, P. (2006). Paimei - reverse engineering framework. 13, 35, 67

Amini, P. and Portnoy, A. (2007). *Sulley: Fuzzing Framework*. 12

Apple (2008). *AppleScript Language Guide*. 13

Apple (2009). Mac os x abi mach-o file format reference. 28, 29

Beck, K. (1999). Embracing change with extreme programming. *Computer*, 32(10):70 –77. 1

Beizer, B. (1990). *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA. 3, 10

Ben-Kiki, O., Evans, C., and dot Net, I. (2009). *YAML Aint Markup Language (YAML) Version 1.2*. 35

Boehm, B. (1986). A spiral model of software development and enhancement. *SIGSOFT Softw. Eng. Notes*, 11(4):14–24. 1

Bruening, D. L. (2004). *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Cambridge, MA, USA. AAI0807735. 5

Campana, G. (2009). Fuzzgrind: an automatic fuzzing tool. Technical report. 6, 23

Chess, B., McGraw, G., and Migues, S. (2010). *BSIMM2: Building Security In Maturity Model.* 9

Chikofsky, E. J. and Cross, J. H. (1990). Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7:13–17. 25, 26

Chodorow, K. and Dirolf, M. (2010). *MongoDB - The Definitive Guide: Powerful and Scalable Data Storage.* O'Reilly. 88

Conger, D., Srinivasamurthy, K., and Cooper, R. (2010). Distributed file fuzzing. U.S. Patent #7,743,281 issued 6/22/2010. 7, 9, 83, 84, 100

Coverity, I. (2010). `http://www.coverity.com`. 4

De Jong, K. A. (1975). *An analysis of the behavior of a class of genetic adaptive systems.* PhD thesis, Ann Arbor, MI, USA. AAI7609381. 58

DeMott, J., Enbody, D. R., and Punch, D. W. F. (2007). Revolutionizing the field of grey-box attack surface revolutionizing the field of grey-box attack surface testing with evolutionary fuzzing. 4, 6, 22, 97

Drewry, W. and Ormandy, T. (2007). Flayer: exposing application internals. In *WOOT '07: Proceedings of the first USENIX workshop on Offensive Technologies*, pages 1–9, Berkeley, CA, USA. USENIX Association. 23

Eagle, C. (2008). *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler.* no starch press, San Francisco, CA, USA. 20, 27, 32, 34, 45, 67, 68, 69

Eddington, M. (2006). Peach fuzzing platform. Electronic. 12, 18

Eddington, M. (2008). Advanced fuzzing with peach 2. 17

Eshelman, L. J. (1990). The chc adaptive search algorithm: How to have safe search when engaging in nontraditional genetic recombination. In Rawlins, G. J. E., editor, *FOGA*, pages 265–283. Morgan Kaufmann. 60, 61

Fogel, L., Owens, A., and Walsh, M. (1966). *Artificial intelligence through simulated evolution.* Wiley, Chichester, WS, UK. 5

Fortify Software, I. (2010). Electronic: `https://www.fortify.com`. 4

Ganesh, V. and Dill, D. L. (2007). A decision procedure for bit-vectors and arrays. In Damm, W. and Hermanns, H., editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 519–531. Springer. 23

Godefroid, P., Levin, M. Y., and Molnar, D. A. (2008). Automated whitebox fuzz testing. In *NDSS*. The Internet Society. 23

Group, A. W. (2011). *AMQP v1.0 revision 0*. AMQP Working Group. 85, 86

Holland, J. H. (1992). *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, Cambridge, MA, USA. 54, 56, 64

Hotchkies, C. (2008). Under the ihood. 33

Intel (1999). *Intel Architecture Software Developer's Manual: Instruction Set Reference Manual*. Intel, Inc. 26

Iozzo, V. (2009). Let your mach-o fly. 29

Johnson, N. and Miller, C. (2010). Crash analysis using bitblaze. 14

Kaukonen, K. and Thayer, R. (1999). A stream cipher encryption algorithm "arcfour". Electronic. 43

keung Luk, C., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Janapa, V., and Hazelwood, R. K. (2005). Pin: Building customized program analysis tools with dynamic instrumentation. In *In Programming Language Design and Implementation*, pages 190–200. ACM Press. 5

Klocwork, I. (2010). Electronic: `http://www.klocwork.com`. 4

Koza, J. R. (1990). Genetic programming: a paradigm for genetically breeding populations of computer programs to solve problems. Technical report, Stanford, CA, USA. 5

Linn, C., Debraydepartment, S., and Science, C. (2003). Obfuscation of executable code to improve resistance to static disassembly. In *In ACM Conference on Computer and Communications Security (CCS)*, pages 290–299. ACM Press. 27

Matz, M., Hubicka, J., Jaeger, A., and Mitchell, M. (2010a). System v application binary interface: Amd64 architecture process supplement. 67

Matz, M., Hubicka, J., Jaeger, A., and Mitchell, M. (2010b). *System V Application Binary Interface: AMD64 Architecture Processor Supplement*. 32, 34

McCabe, T. J. (1976). A complexity measure. In *Proceedings of the 2nd international conference on Software engineering*, ICSE '76, pages 407–, Los Alamitos, CA, USA. IEEE Computer Society Press. 69, 70, 93

Miller, B. P., Fredriksen, L., and So, B. (1990). An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44. 10, 28

Miller, C. (2010). Babysitting an army of monkeys - fuzzing 4 products with 5 lines of python. Electronic: `http://securityevaluators.com/files/slides/cmiller_CSW_2010.ppt`. 8

Miller, C. and Zovi, D. D. (2009). *The Mac Hacker's Handbook*. Wiley Publishing. 32

Milller, B. (1988). Cs736 project list. Electronic. 10

Mitchell, M. (1998). *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, USA. 55, 56, 58

Mitre (2010). Cve-2010-1120. Electronic. 77

Molnar, D. and Wagner, D. (2007). Catchconv: Symbolic execution and run-time type inference for integer conversion errors. Technical report, UC Berkeley EECS. 6

Molnar, D. A. (2009). *Dynamic Test Generation for Large Binary Programs*. PhD thesis, EECS Department, University of California, Berkeley. 24

Monti, E., Rohlf, C., and Tracy, M. (2009). Ruby for pentesters. 13

Nethercote, N. (2004). *Dynamic Binary Analysis and Instrumentation*. PhD thesis, Trinity College, Cambridge, UK. 5, 44, 45, 77, 78

Nethercote, N. and Seward, J. (2007). Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)*, pages 89–100, San Diego, California, USA. 45, 47

Neystadt, J. (2008). Automated penetration testing with white-box fuzzing. Electronic: `http://msdn.microsoft.com/en-us/library/cc162782.aspx`. 9

Prowell, S. (1998). Impact of sequence-based software specification on statistical software testing. In *Proceedings of the Second International Software Quality Week Europe*, San Francisco, California. Software Research Institute, Inc., Software Research Institute, Inc. 2

Prowell, S. J. and Poore, J. H. (2003). Foundations of sequence-based software specification. *IEEE Transactions on Software Engineering*, 29:417–429. 2

Rechenberg, I. (1971). *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. PhD thesis, TU Berlin. 5

Schneier, B. (1995). *Applied cryptography (2nd ed.): protocols, algorithms, and source code in C*. John Wiley & Sons, Inc., New York, NY, USA. 43

Schwarz, B., Debray, S., and Andrews, G. (2002). Disassembly of executable code revisited. In *In Proc. IEEE 2002 Working Conference on Reverse Engineering (WCRE)*, pages 45–54. IEEE Computer Society. 27

Shacham, H., jin Goh, E., Modadugu, N., Pfaff, B., and Boneh, D. (2004). On the effectiveness of address-space randomization. In *In CCS 04: Proceedings of the 11th ACM Conference on Computer and Communications Security*, pages 298–307. ACM Press. 33

Shaw, Z. (2006). Rfuzz. Electronic. 43

Sivanandam, S. and Deepa, S. (2007a). *Introduction to Genetic Algorithms*. Springer, New York, NY, USA. 5, 82, 99

Sivanandam, S. N. and Deepa, S. N. (2007b). *Introduction to Genetic Algorithms*. Springer Publishing Company, Incorporated, 1st edition. 55, 57, 58

Sparks, S., Embleton, S., Cunningham, R., and Zou, C. C. (2007). Automated vulnerability analysis: Leveraging control flow for evolutionary input crafting. In *ACSAC*, pages 477–486. 4, 6, 20, 21, 70, 71, 72, 73

Sthamer, H., Baresel, A., and Wegener, J. (2001). Evolutionary testing of embedded systems. In *14th International Internet and Software Quality Week*. 22

Sutton, M. and Greene, A. (2005). The art of file format fuzzing. Electronic: `http://www.blackhat.com/presentations/bh-jp-05/bh-jp-05-sutton-greene.pdf`. 8, 15, 16, 17

Sutton, M., Greene, A., and Amini, P. (2007). *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional. 10, 11

Sweetman, D. (2006). *See MIPS Run, Second Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. 26

Takanen, A., DeMott, J., and Miller, C. (2008). *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, Inc., Norwood, MA, USA. 2, 11, 12, 22, 75

Tanenbaum, A. S. and Goodman, J. R. (1998). *Structured Computer Organization*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 4th edition. 80, 81

van Sprundel, I. (2005). Fuzzing. Chaos Computer Club. 15

Vose, M. D. (1998). *The Simple Genetic Algorithm: Foundations and Theory.* MIT Press, Cambridge, MA, USA.

Vuagnoux, M. (2005). Autodafe: an act of software torture. Chaos Computer Club. 19, 20, 59

Wegener, J. (2001). Overview of evolutionary testing. Electronic. 22

Appendices

Appendix A: Abbreviations

ABI Application Binary Interface

AMQP Advanced Message Queuing Protocol

ANTLR ANother Tool for Language Recognition

API Application Programming Interface

ASLR Address Space Layout Randomization

BGA Byte Genetic Algorithm

BSON Binary JSON

BSS Block Started by Symbol

CHC-GA Cross-Generational Elitist Selection, Heterogeneous Recombination, and Cataclysmic Mutation Genetic Algorithm

CLI Command-line Interface

DBI Dynamic Binary Instrumentation Framework

DoS Denial Of Service

ELF Executable and Linkable Format

EFS Evolutionary Fuzzing System

FLIRT Fast Library Identification and Recognition Technology

GCC GNU Compiler Collection

GNU GNU's Not Unix!

GPL GNU Public License

GUI Graphical User Interface

HDFS Hadoop Distributed File System

HUX Half Uniform Crossover

IP Internet Protocol

IS-IS Intermediate System-to-Intermediate System

IR Intermediate Representation

JIT just-in-time

Mach-O Mach Object

MIPS32 Microprocessor without Interlocked Pipeline Stages

PDF Portable Document Format

PE Portable Executable

RISC Reduced Instruction Set

RFC Request For Comments

SDK Software Development Kit

SGA Simple Genetic Algorithm

SGA-MIP Simple Genetic Algorithm with Mangled Initial Population

SGA-MM Simple Genetic Algorithm Mangle Mutator

SNMP Simple Network Management Protocol

TLV Type-length-value

TIFF Tagged Image File Format

SWIG Simplified Wrapper Interface Generator

UTF8 UCS Transformation Format 8-bit

UX Uniform Crossover

UUID Universally Unique Identifier

XML Extensible Markup Language

YAML YAML Ain't Markup Language

Appendix B: Code Examples

```
#!/bin/bash

#
# Usage: ./fuzz.sh input.tiff #testcases program
#

#
# Make sure testcase directory exists
#
if [ ! -e testcases ] ; then
    mkdir testcases
fi

#
# Check for crashes directory
#
if [ ! -e crashes ] ; then
    mkdir crashes
fi

#
# Make sure original test case exists
#
if [ ! -e $1 ] ; then
    echo "Missing base case: $1"
    exit 1
fi

#
# Loop for specified number of test cases
#
for i in $(seq $2) ; do
    echo "Running Testcase: $i"

    #
    # Generate the testcase
    #
    cp $1 testcases/$1.$i
    ./mangle testcases/$1.$i

    #
    # Just running tiffinfo for now
    #
    $3 testcases/$1.$i > /dev/null 2>&1

    #
    # save return variable
    #
    returnVar=$?
    sleep 2

    #
    # Check for crash (SIGILL, SIGABRT, SIGBUS, SIGSEGV (128 + signal number))
    #
    if [ $returnVar -eq 132 -o $returnVar -eq 134 -o $returnVar -eq 138 -o ↩
        $returnVar -eq 139 ] ; then
        mv testcases/$1.$i crashes/
    fi

    #
    # Kill outstanding processes if any
    #
    progPid=`ps aux | grep $3 | grep -v grep | awk '{print $2}'`
    kill -9 $progPid > /dev/null 2>&1
done

exit 0
```

Figure B.1: Mangle Wrapper Script

```
s_block_start("imageFileHdr");
s_binary("0x49 0x49");                   //Endianness
s_binary("0x2A 0x00");                   //Magic Number
s_binary("0x3c 0x00 0x00 0x00");         //Image File Directory pointer
s_block_end("imageFileHdr");

s_block_start("dataSection");
s_binary("ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff↩
    ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff dd cc bb ↩
   aa");
s_block_end("dataSection");
s_block_start("imageFileDirectory");
s_binary("0x03 0x00");
s_block_start("imageLengthDirectoryEntry");
s_binary("0x01 0x01 0x03 0x00 0x01 0x00 0x00 0x00 0x19 0x00 0x00 0x00");
s_block_end("imageLengthDirectoryEntry");
s_block_start("bitsPerSampleDirectoryEntry");
s_binary("0x02 0x01 0x03 0x00");
s_int_variable(30, 9);              //bitsPerSample Entry Number of Entries
s_binary("0x08 0x00 0x00 0x00");
s_block_end("bitsPerSampleDirectoryEntry");
s_block_start("stripOffsetsDirectoryEntry");
s_binary("0x11 0x01 0x04 0x00 0x01 0x00 0x00 0x00 0x08 0x00 0x00 0x00");
s_block_end("stripOffsetsDirectoryEntry");
s_binary("0x00 0x00");              //Pointer to other Image File Directory
s_block_end("imageFileDirectory");
```

Figure B.2: notSPIKEfile Data Model

```
<?xml version="1.0" encoding="utf-8"?>
<Peach xmlns="http://phed.org/2008/Peach" xmlns:xsi="http://www.w3.org/2001/↩
    XMLSchema-instance"
        xsi:schemaLocation="http://phed.org/2008/Peach /peach/peach.xsd">

<Include ns="default" src="file:defaults.xml" />

<DataModel name="imageFileHdr">
    <Number size="16" endian="little" value="18761"/>
    <Number size="16" endian="little" value="42"/>
    <Number size="32" endian="little" value="60"/>
</DataModel>

<DataModel name="imageDirectoryEntry">
    <Number size="16" endian="little"/>
    <Number size="16" endian="little"/>
    <Number size="32" endian="little"/>
    <Number size="32" endian="little"/>
</DataModel>

<DataModel name="imageFileDirectory">
    <Number size="16" endian="little" value="3"/>
    <Choice minOccurs="3" maxOccurs="5">
        <Block ref="imageDirectoryEntry"/>
    </Choice>
    <Number size="32" endian="little" value="0"/>
</DataModel>

<!-- Defines the format of the whole TIFF file -->
<DataModel name="TIFF">
    <Block ref="imageFileHdr"/>
    <Blob name="dataSection" valueType="hex" value="ff ff ff ff ff ff ff ff ff ff ff ff↩
         ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ↩
        ff ff ff ff ff ff ff ff ff dd cc bb aa"/>
    <Block ref="imageFileDirectory"/>
</DataModel>

<!-- DataModel for the parameter (fuzzed file) passed to tiffinfo -->
<DataModel name="Param">
    <String name="Value" isStatic="true"/>
</DataModel>

<!-- State Model for what the fuzzer does -->
<StateModel name="TIFFStates" initialState="Initial">
    <State name="Initial">
        <!-- Write out the Tiff file -->
        <Action type="output">
            <!-- Refer to the tiff model -->
            <DataModel ref="TIFF"/>
        </Action>
    </State>
</StateModel>

    <Test name="TIFFTest">
        <StateModel ref="TIFFStates"/>

        <!-- Publisher (Write all the files) -->
        <Publisher class="file.FilePerIteration">
            <Param name="filename" value="fuzzed-%d.tiff"/>
        </Publisher>
    </Test>

    <!-- Configure a single run -->
    <Run name="DefaultRun">
        <Logger class="logger.Filesystem">
            <Param name="path" value="logs"/>
        </Logger>
        <Test ref="TIFFTest" />
    </Run>
</Peach>
```

Figure B.3: Peach XML Configuration

Appendix C: Command Reference

```
distrib
-------
thor distrib:dstart   # Start the Mongodb database for distribution
thor distrib:dstop    # Stop the Mongodb database for distribution
thor distrib:qreset   # Resetting the rabbitmq queueing system
thor distrib:qstart   # Start the rabbitmq queueing system
thor distrib:qstatus  # Query the rabbitmq queueing system
thor distrib:qstop    # Stop the rabbitmq queueing system
thor distrib:start    # Start all distributed fuzzing components
thor distrib:stop     # Stop all distributed fuzzing components

fuzz
----
thor fuzz:package    # Package Fuzzer Configuration Files
thor fuzz:report     # Print a report of current activity to the logfile
thor fuzz:start      # Start a Mamba Fuzzer
thor fuzz:stop       # Stop a Mamba Fuzzer
thor fuzz:unpackage  # Unpackage Fuzzer Configuration Files

tools
-----
thor tools:disassemble  # Extract disassembly information from IDA Pro into Y...
thor tools:otool        # Runs otool recursively over an object to display li...
thor tools:seed         # Seed test cases from google searches
```

Figure C.1: Mamba Fuzzing Framework Command Listing

```
Usage:
  thor tools:otool

Options:
  -o, [--object=OBJECT]  # Executable or shared library to examine with otool
  -a, [--architecture]   # Turn 64 bit support on

Runs otool recursively over an object to display linked objects
```

Figure C.2: Mamba tools:otool Command Reference

```
Usage:
  thor tools:disassemble

Options:
  -o, [--object=OBJECT]  # Executable or shared library to disassembly and extract
  -a, [--architecture]   # Turn 64 bit support on
  -i, [--ida=IDA]        # Set a different IDA Pro path

Extract disassembly information from IDA Pro into YAML serialization format
```

Figure C.3: Mamba tools:disassemble Command Reference

```
Usage:
  mamba create

Options:
  -a, [--app=APP]              # Path to an executable (ex. /Applications/Preview.app/↩
      Contents/MacOS/Preview)
                               # Default: /Applications/Preview.app/Contents/MacOS/↩
                                  Preview
  -d, [--distributed]          # Set the fuzzer type as distributed
  -e, [--executor=EXECUTOR]    # Type of executor to use (cli, appscript)
                               # Default: appscript
  -p, [--port=N]               # Port for distributed fuzzing
                               # Default: 27017
  -t, [--timeout=N]            # Time to run an executable for each testcase in ↩
      seconds (ex. 20)
                               # Default: 0
  -s, [--server=SERVER]        # Server address for distributed fuzzing (ex. fuzz.io)
                               # Default: fuzz.io
  -y, [--type=TYPE]            # Type of fuzzer to create (Supported Types: [”↩
      ByteGeneticAlgorithm”, ”CHCGeneticAlgorithm”, ”Mangle”, ”↩
      MangleGeneticAlgorithm”, ”SimpleGeneticAlgorithm”])
                               # Default: Mangle

Create a Mamba instance
```

Figure C.4: Mamba create Command Reference

```
Usage:
  thor tools:seed

Options:
  -c, [--clean]                # Cleanup downloaded files, since they are zipped
  -f, [--filetype=FILETYPE]    # Filetype to download examples (ex. pdf, doc, ppt)
                               # Default: pdf
  -n, [--num=N]                # Number of files to download
                               # Default: 10
  -t, [--terms=TERMS]          # File containing search terms to mix into test case ↩
      seeding
  -z, [--zip]                  # Zip the test cases downloaded by the google crawler

Seed test cases from google searches
```

Figure C.5: Mamba tools:seed Command Reference

```
Usage:
  thor fuzz:package

Package Fuzzer Configuration Files
```

Figure C.6: Mamba fuzz:package Command Reference

```
Usage:
  thor fuzz:unpackage

  Unpackage Fuzzer Configuration Files
```

Figure C.7: Mamba fuzz:unpackage Command Reference

```
Usage:
  thor distrib:qstart

Start the rabbitmq queueing system
```

Figure C.8: Mamba distrib:qstart Command Reference

```
Usage:
  thor distrib:qstop

Stop the rabbitmq queueing system
```

Figure C.9: Mamba distrib:qstop Command Reference

```
Usage:
  thor distrib:qreset

  Resetting the rabbitmq queueing system
```

Figure C.10: Mamba distrib:qreset Command Reference

```
Usage:
  thor distrib:qstatus

Query the rabbitmq queueing system
```

Figure C.11: Mamba distrib:qstatus Command Reference

```
Usage:
  thor distrib:dstart

Start the Mongodb database for distribution
```

Figure C.12: Mamba distrib:dstart Command Reference

```
Usage:
  thor distrib:dstop

Stop the Mongodb database for distribution
```

Figure C.13: Mamba distrib:dstop Command Reference

```
Usage:
  thor distrib:start

Start all distributed fuzzing components
```

Figure C.14: Mamba distrib:start Command Reference

```
Usage:
  thor distrib:stop

Stop all distributed fuzzing components
```

Figure C.15: Mamba distrib:stop Command Reference

Appendix D: Graphs

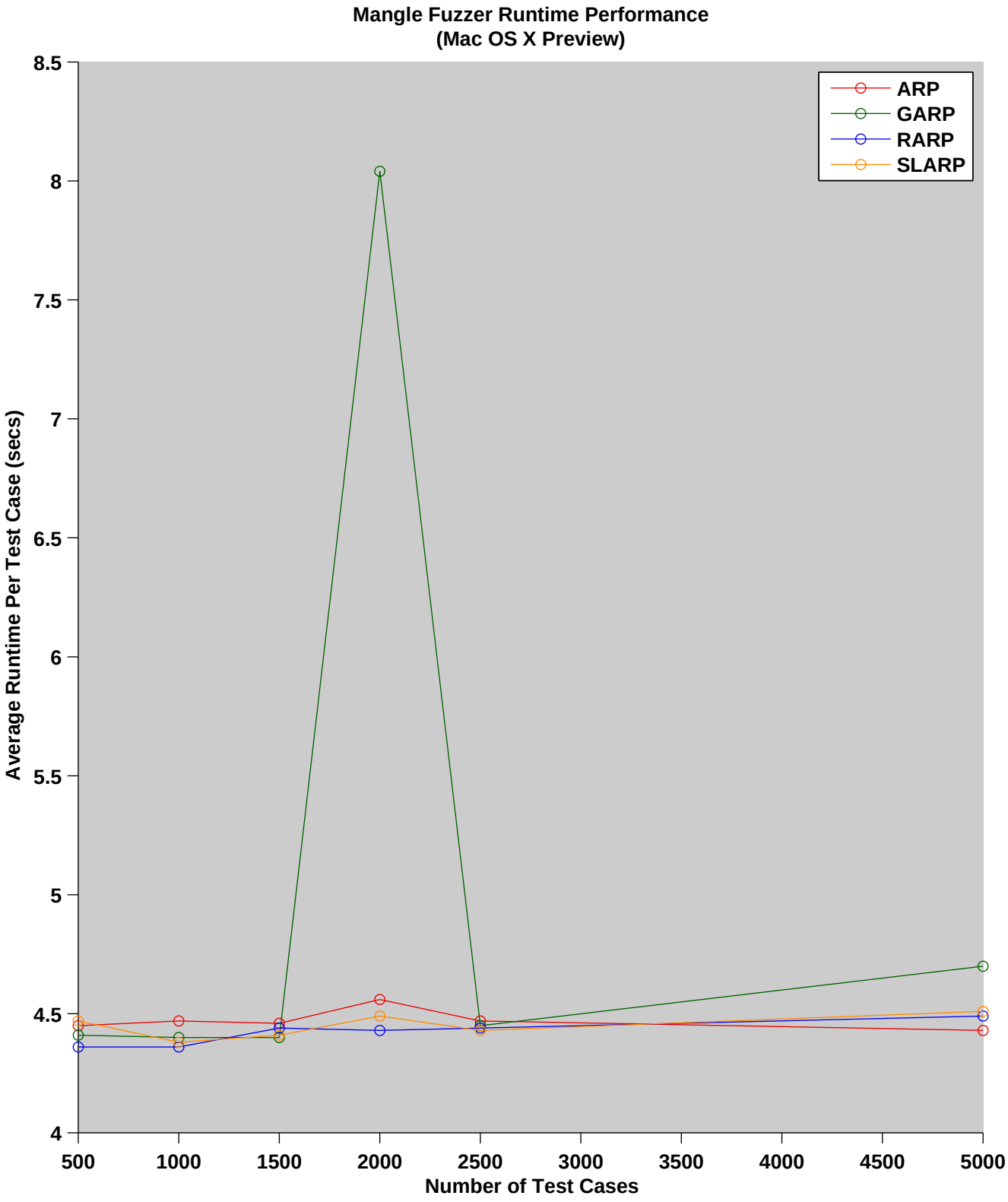


Figure D.1: Mangle Fuzzer Performance

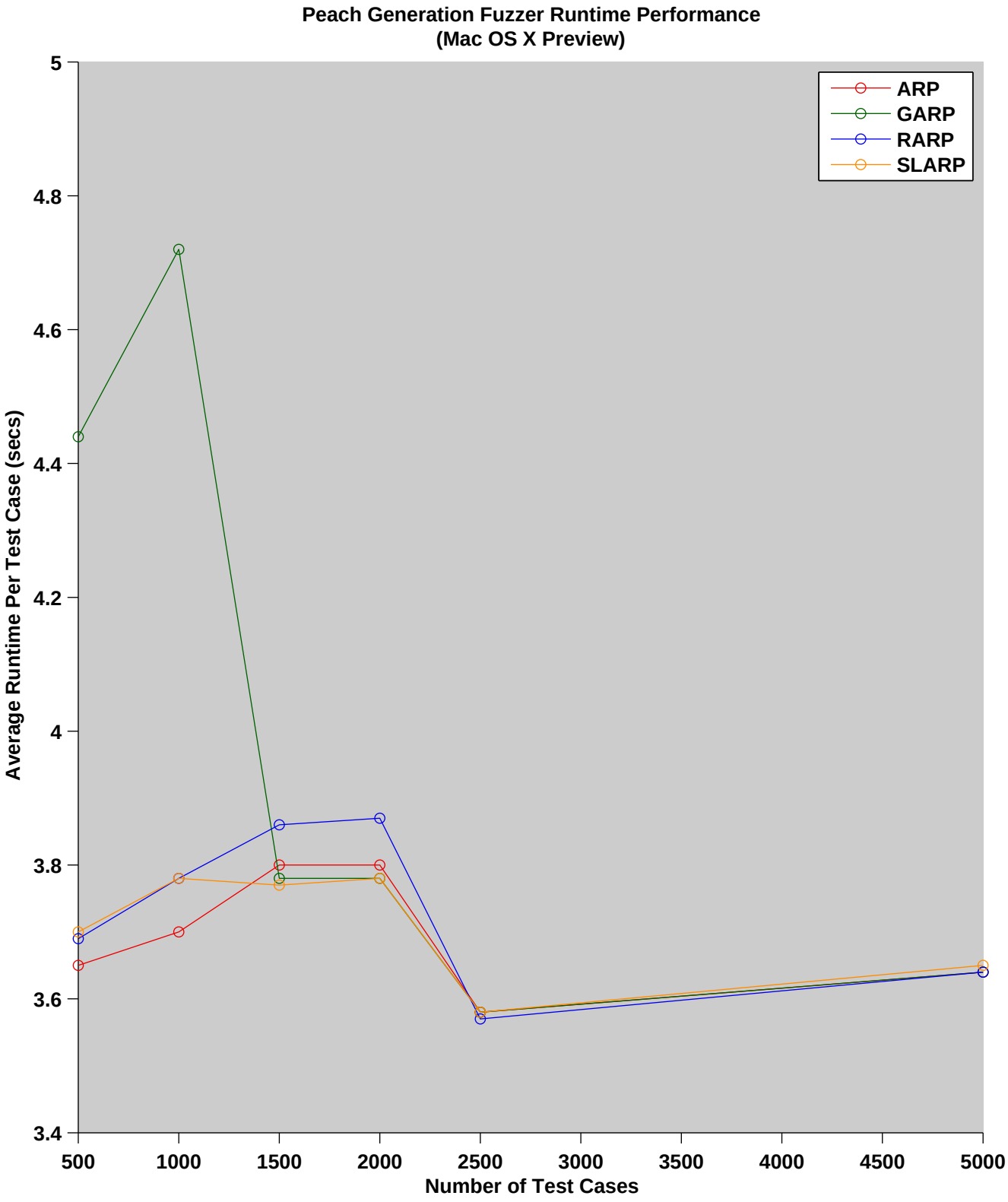


Figure D.2: Peach Fuzzer Performance

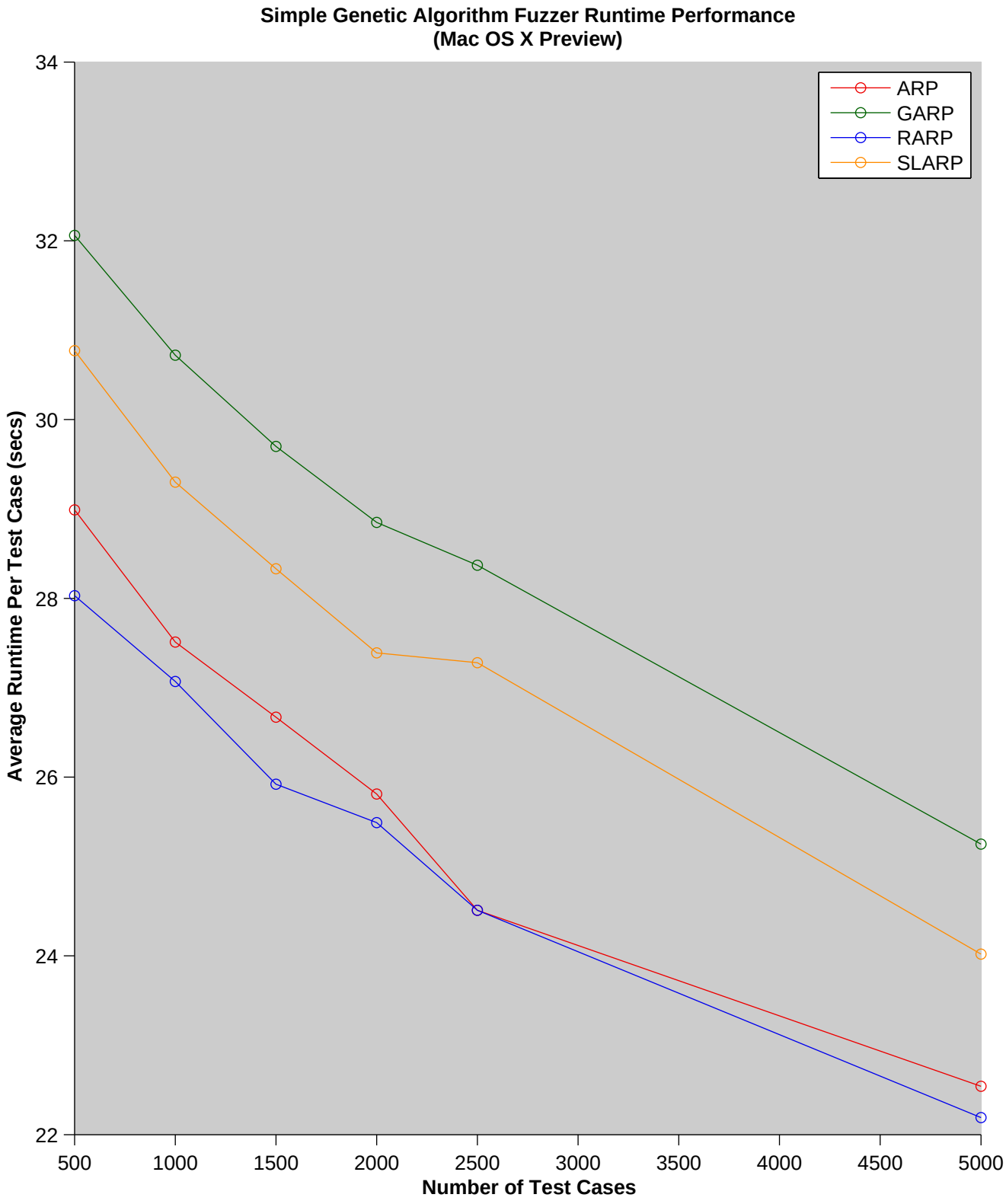


Figure D.3: Simple Genetic Algorithm Fuzzer Performance

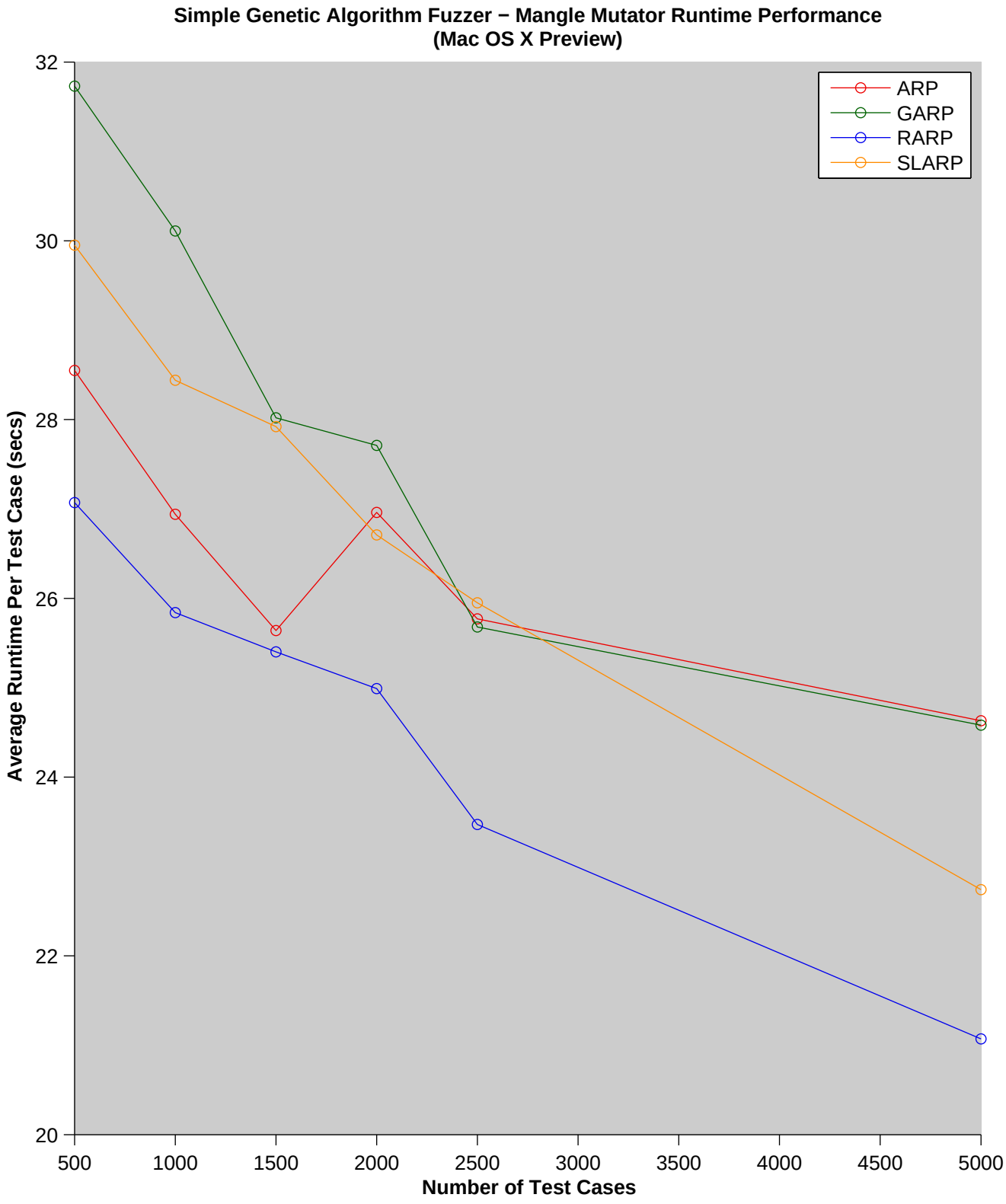


Figure D.4: Simple Genetic Algorithm Fuzzer with Mangle Mutator Performance

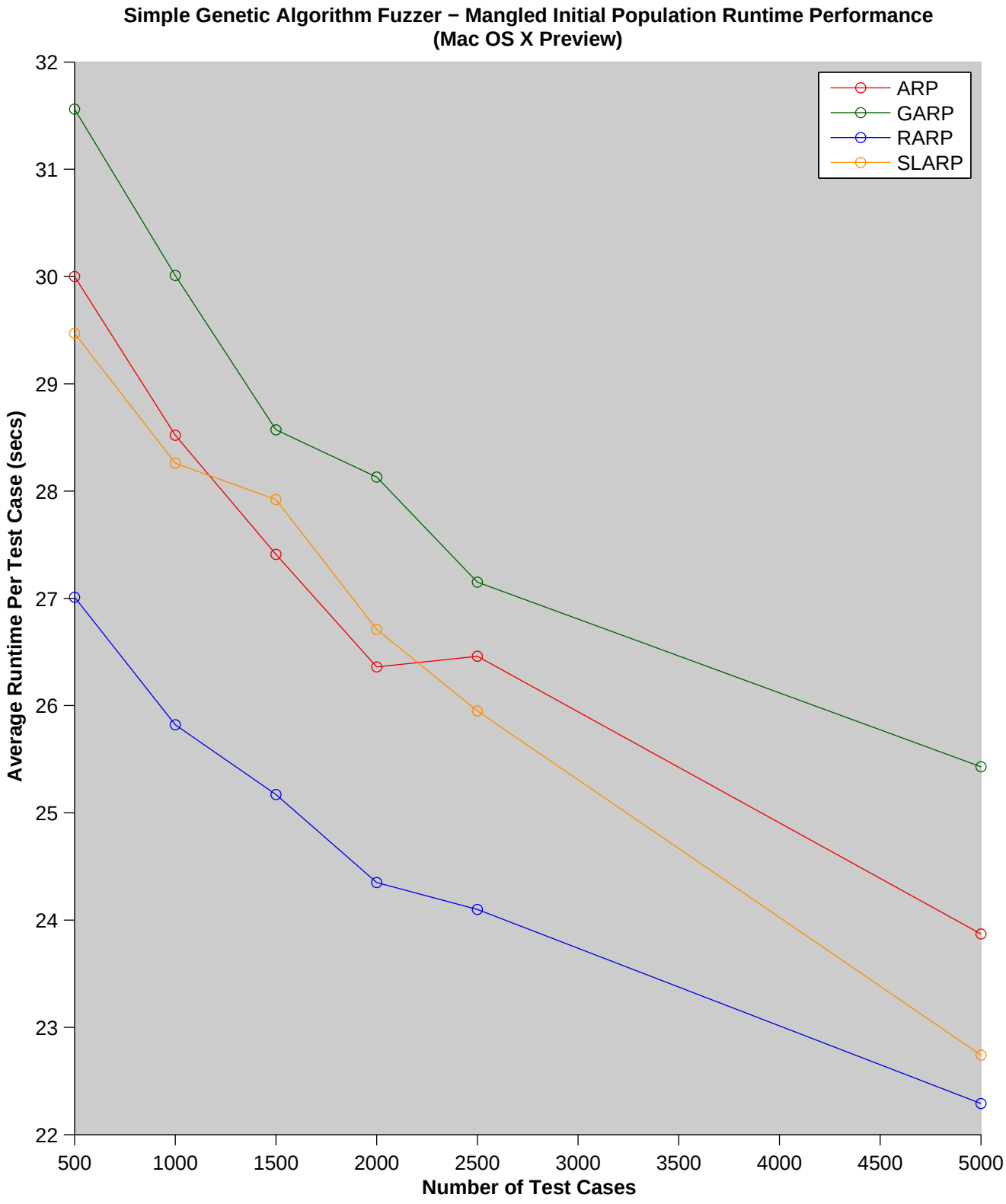


Figure D.5: Simple Genetic Algorithm Fuzzer with Mangled Initial Population Performance

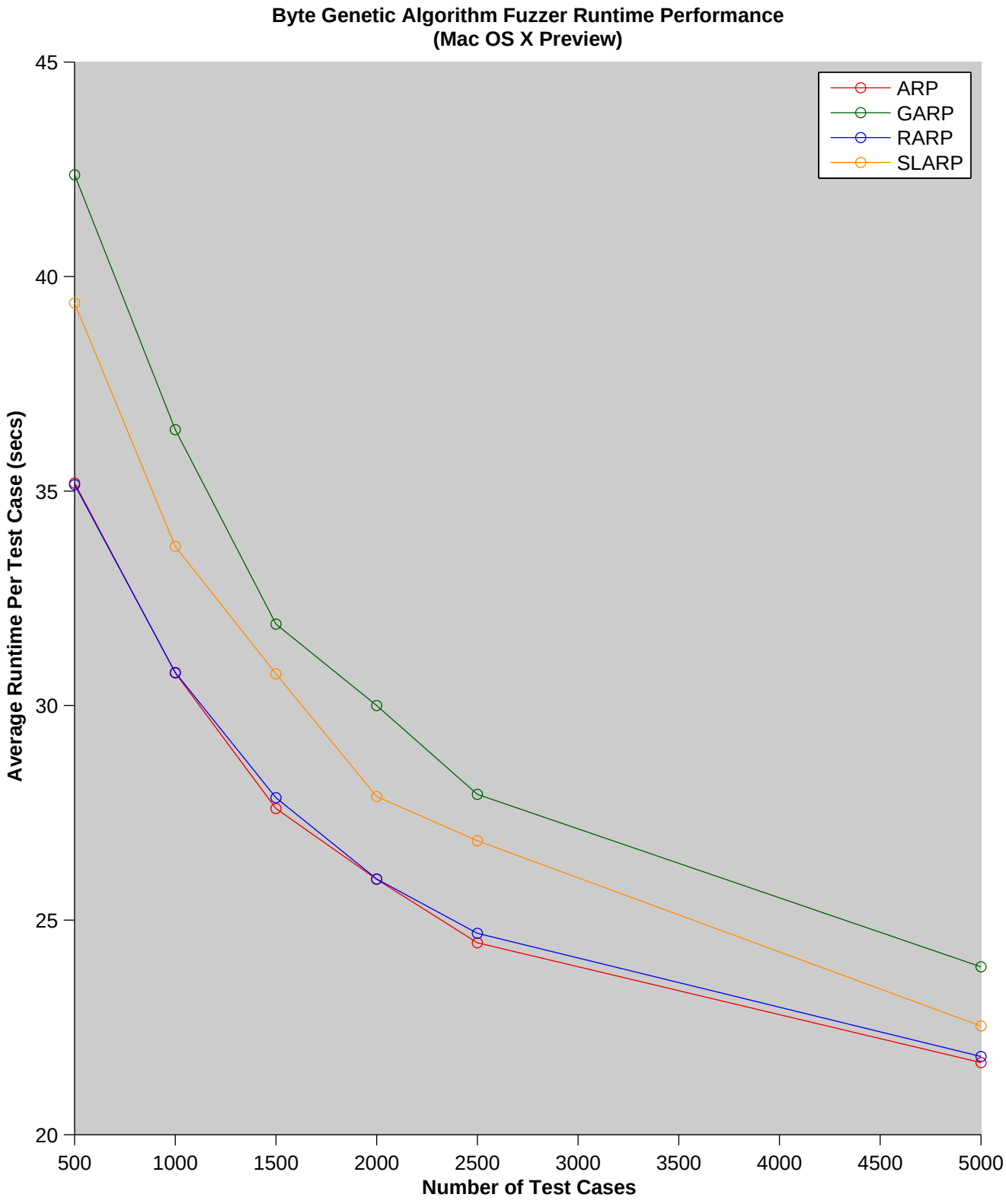


Figure D.6: Byte Genetic Algorithm Fuzzer Performance

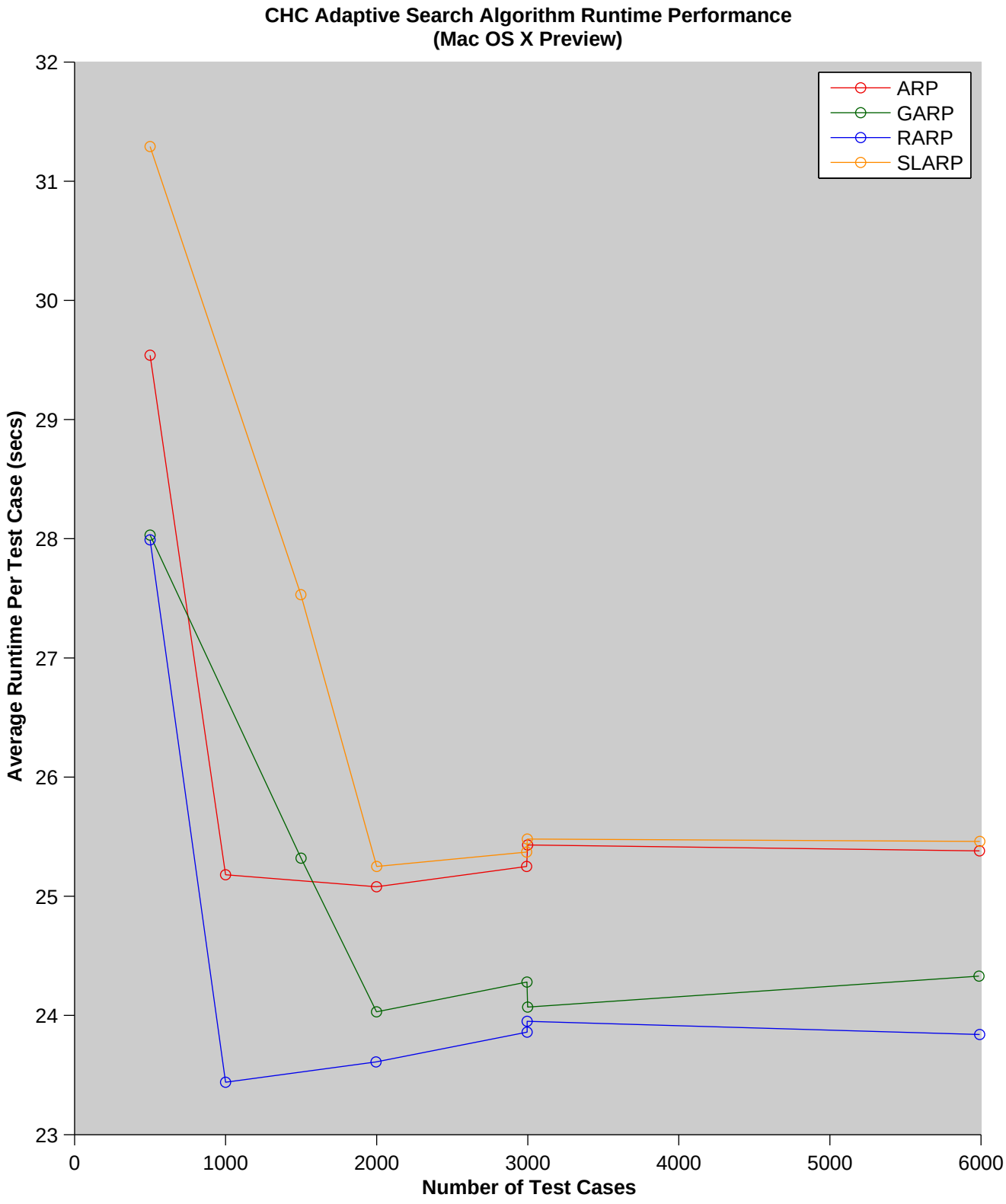


Figure D.7: CHC Genetic Algorithm Fuzzer Performance

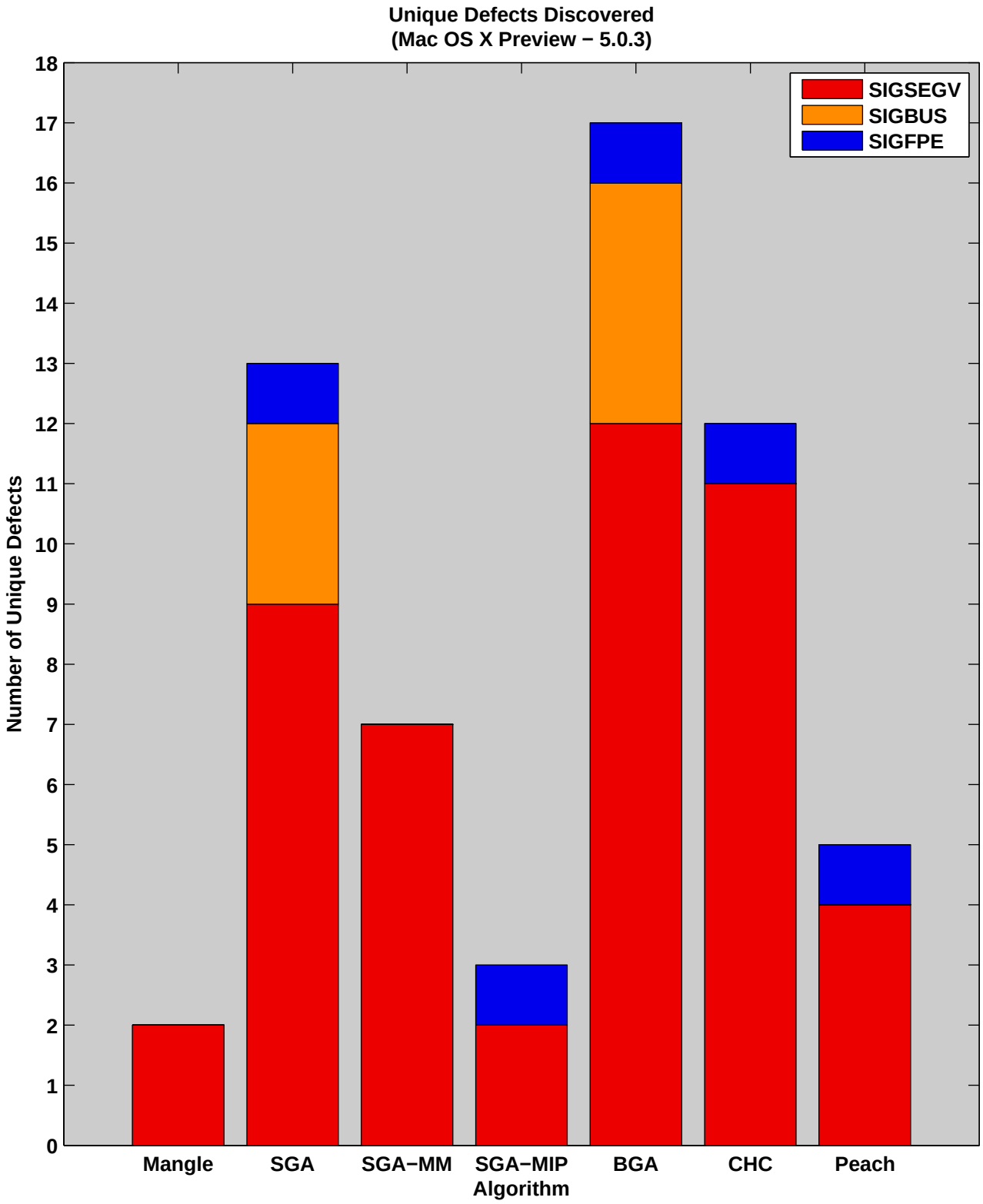


Figure D.8: Defects Found per Fuzzing Algorithm

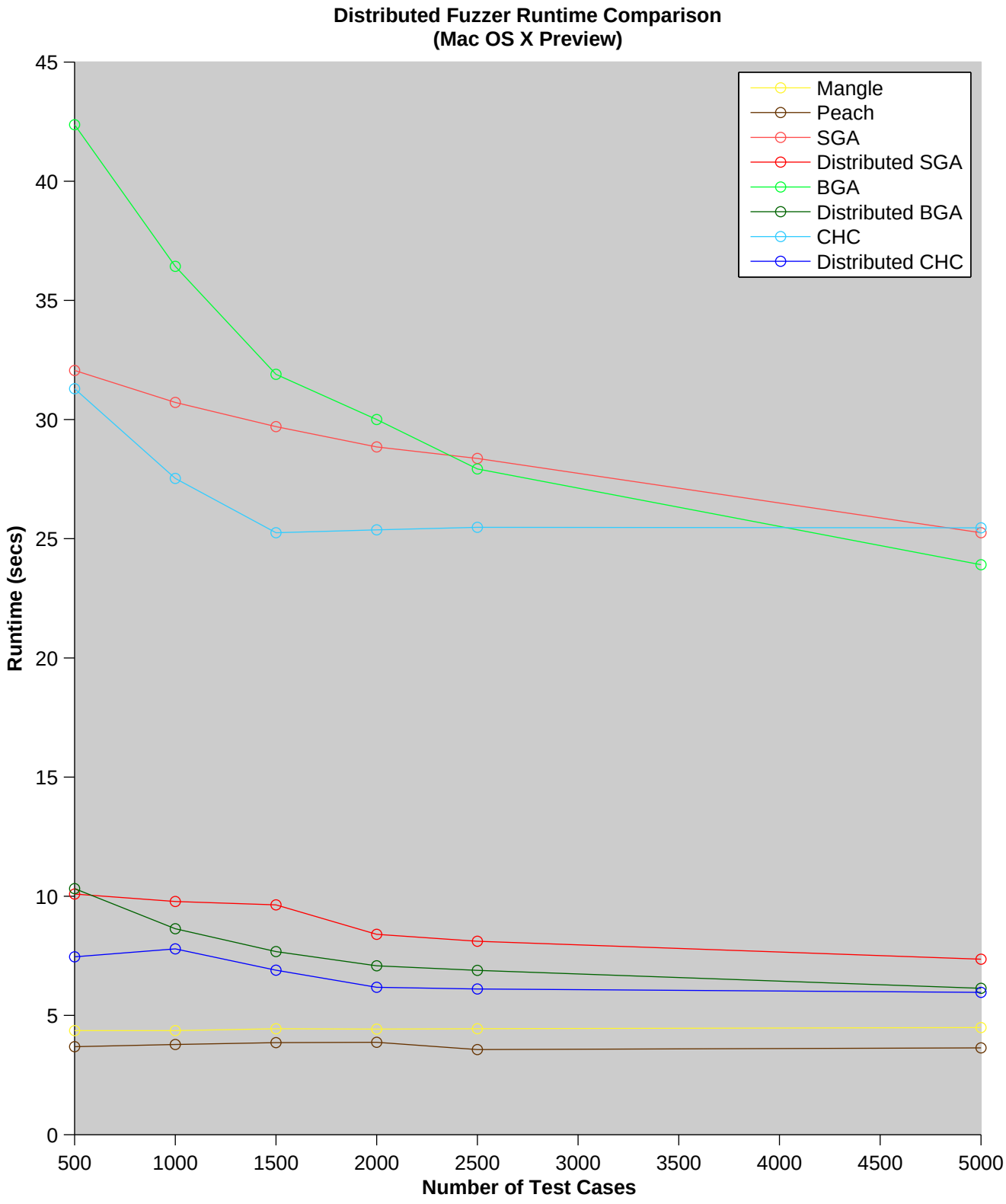


Figure D.9: Performance Comparison with Distributed Environment

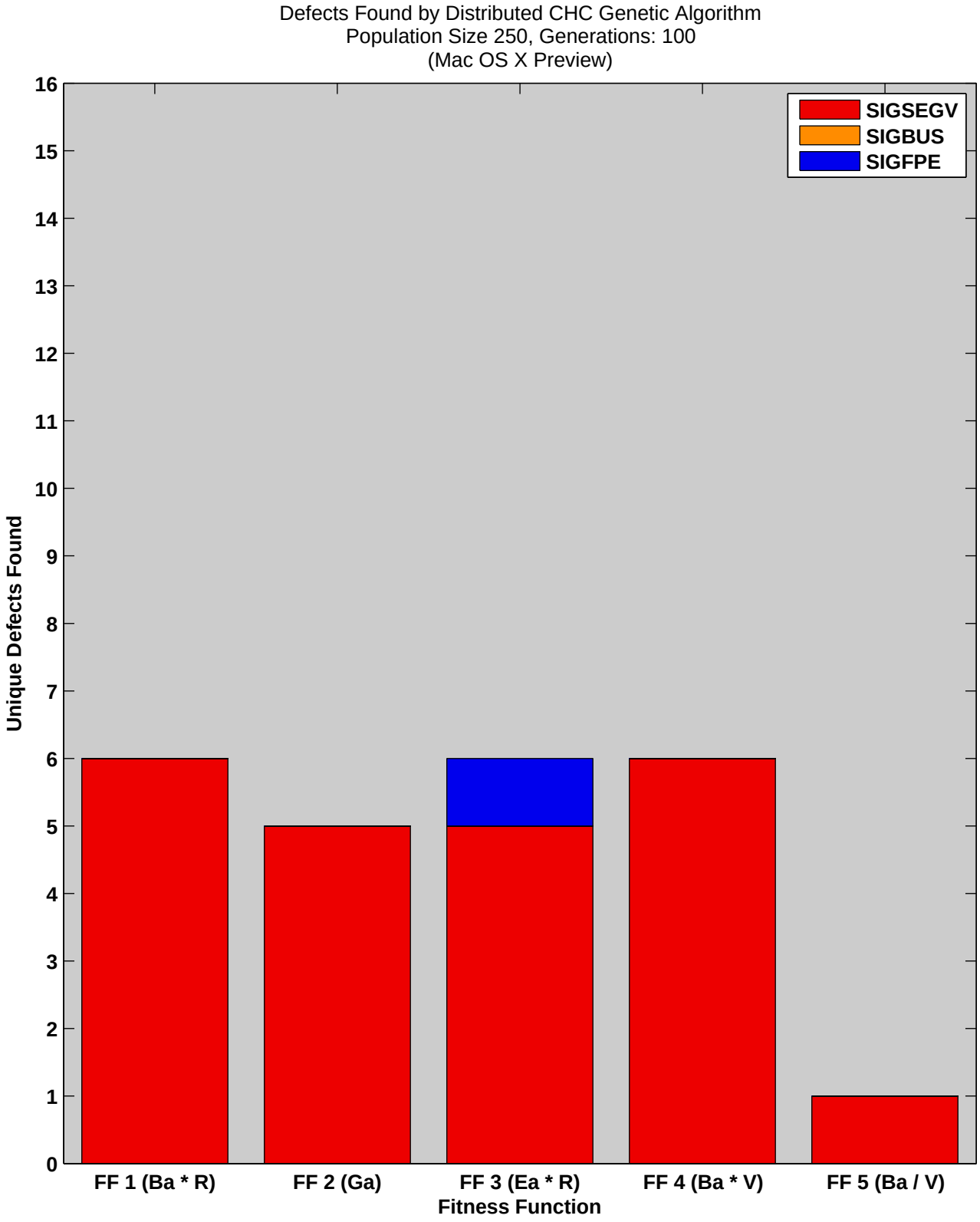


Figure D.10: Distributed CHC GA Unique Defects Found per Fitness Function (250)

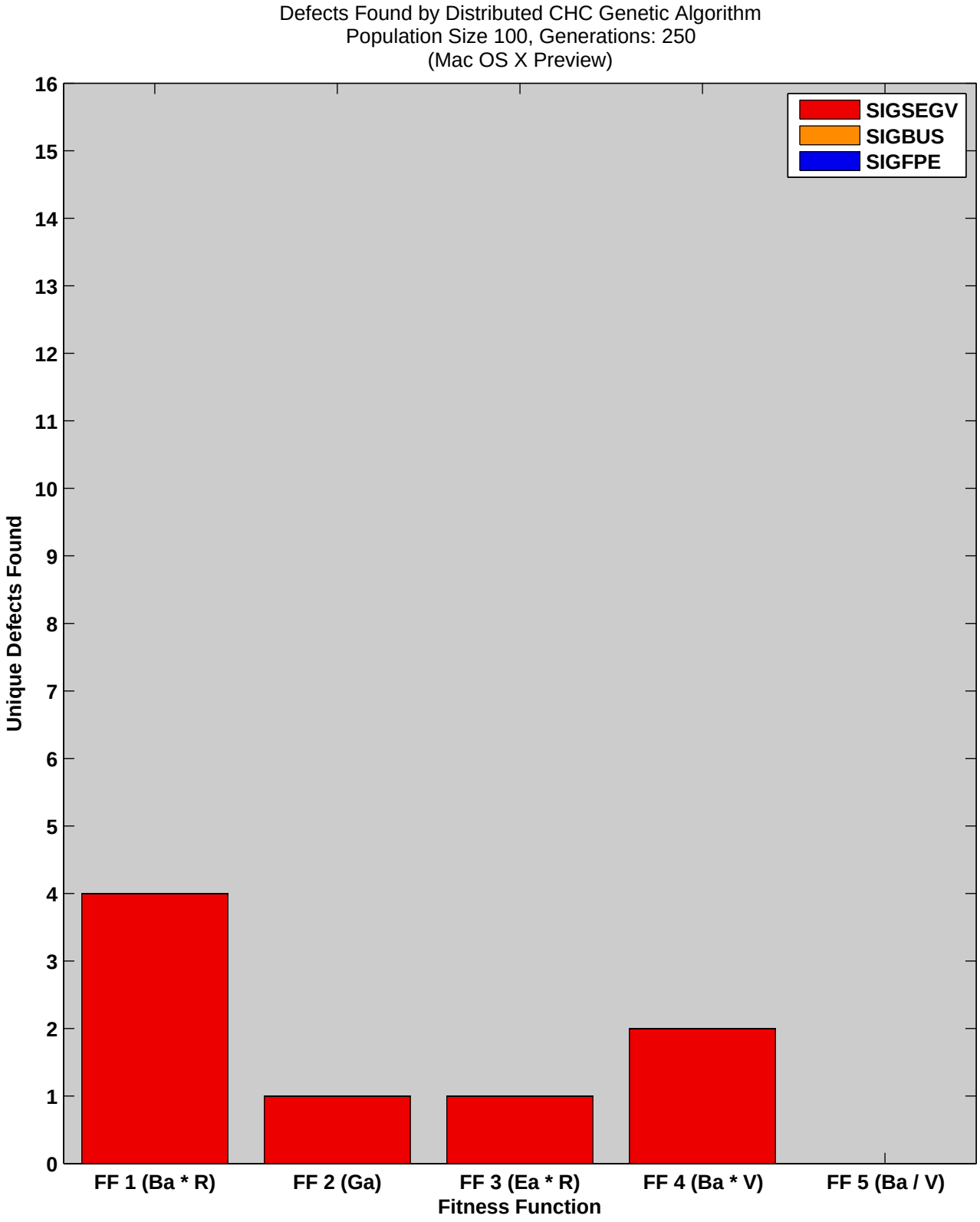


Figure D.11: Distributed CHC GA Unique Defects Found per Fitness Function (100)

Vita

Roger Seagle, Jr. was born on March 24, 1981 in Wilmington, NC. He spent most of his childhood in the foothills town of Morganton, NC. In Fall 1999, he attended Wake Forest University and subsequently graudated with a degree in Computer Science in the Spring of 2003. Enamoured by the problems within Computer Science, he continued his education towards a doctoral degree at the University of Tenessee, Knoxville in the Fall of 2003. During his initial two and a half year stint at the university, Roger served a term as ACM Vice President and ACM President along with performing duties as a Teaching Assitant and Research Assistant. He completed a Masters degree in Computer Science in the Winter of 2005 and pursued employment as a sotware engineer at Cisco Systems, Inc. Over the next few years, he maintained software for routing/switching and other telecommunications devices. In Fall 2009, he re-entered the University of Tennessee, Knoxville to finish his doctoral research and eventually completed his Doctor of Philosophy degree in Fall 2011. Currently, he resides in Knoxville, TN and continues working for Cisco Systems, Inc. as a Technical Leader. He is a member of ACM and USENIX, and his research interests are in network architectures, network protocols, computer security, reverse engineering, and evolutionary algorithms.