ROP Gadget Prevalence and Survival under Compiler-based Binary Diversification Schemes

Joel Coffman
The Johns Hopkins University
Applied Physics Laboratory
joel.coffman@jhuapl.edu

Daniel M. Kelly
The Johns Hopkins University
Applied Physics Laboratory
daniel.m.kelly@jhuapl.edu

Christopher C. Wellons
The Johns Hopkins University
Applied Physics Laboratory
christopher.wellons@jhuapl.edu

Andrew S. Gearhart
The Johns Hopkins University
Applied Physics Laboratory
andrew.gearhart@jhuapl.edu

ABSTRACT

Diversity has been suggested as an effective alternative to the current trend in rules-based approaches to cybersecurity. However, little work to date has focused on how various techniques generalize to new attacks. That is, there is no accepted methodology that researchers use to evaluate diversity techniques. Starting with the hypothesis that an attacker's effort increases as the common set of executable code snippets (return-oriented programming (ROP) gadgets) decreases across application variants, we explore how different diversification techniques affect the set of ROP gadgets that is available to an attacker. We show that a small population of diversified variants is sufficient to eliminate 90–99% of ROP gadgets across a collection of real-world applications. Finally, we observe that the number of remaining gadgets may still be sufficient for an attacker to mount an effective attack regardless of the presence of software diversity.

CCS Concepts

•**Security and privacy** → **Systems security;** *Malware and its mitigation;*

Keywords

software diversity; compiler transformations; evaluation; code reuse attacks; return-oriented programming (ROP)

1. INTRODUCTION

The proliferation of mass-market software targeting a single instruction set architecture (ISA) creates security risks: a vulnerability in an application represents a common avenue of attack against all instances of that application [14, 16]. The consequences and impact of such risks have been repeatedly observed through well-known attacks such as the Morris worm, Nimda, and Conficker—malware that collectively spans two decades. This existing hardware and software "monoculture" [16] stands in marked contrast to biological development. Scientists now understand the importance of diversity to a species'—or even to an entire ecosystem's—health and survival against natural catastrophes and even targeted eradication [14, 39]. Applying these same principles to software may reap similar dividends: a successful attack impacts only a subset of the instances of a vulnerable application. This idea is similar to herd immunity in epidemiology [19] where a portion of the population (application instances) is immune to an infection (exploit). Ideally, a given exploit will only be successful against a single application instance even if the underlying vulnerability is shared by many instances.

Such a scenario decidedly shifts the balance of power in cybersecurity. Instead of a compromise-once, compromise-everywhere model where exploits are commodities, exploits become unique to each application instance. As a result, attacks must become more targeted to combat their increased cost. Moreover, existing tiers of attackers (e.g., nuisance attackers that exploit known vulnerabilities [17]) will be eliminated due to the lack of reusable exploits.

The concepts underlying software diversity trace their roots to fault tolerance. Fault-tolerant systems typically use diversity and redundancy to achieve reliability. Diversity minimizes the likelihood that multiple implementations of a given component will produce an erroneous result [2]. Redundancy masks independent errors in these components, typically via a voting mechanism. Both techniques—but particularly diversity—have also been suggested to reduce the cybersecurity risks of a software monoculture. Seminal work [9, 14] led to a wide variety of approaches that introduce diversity into various stages of the software life cycle from source code obfuscation to ISA randomization.

Despite its potential, software diversity is not a cybersecurity panacea [28]. For software diversity to be effective, it must significantly increase the cost to the adversary of conducting a cyber attack. Exploits that rely on fixed characteristics of a binary such as particular sequences of instructions are likely to be impacted by various diversity techniques. For example, disparate implementations of the same application may not share vulnerabilities even if each implementation contains unsafe memory handling. Obfuscated or optimized code may interfere with the exploit, possibly causing the

SPRO'16, October 28 2016, Vienna, Austria
© 2016 ACM. ISBN 978-1-4503-4576-7/16/10...$15.00
DOI: http://dx.doi.org/10.1145/2995306.2995309

application to crash. Even if a vulnerability is shared among application variants, attackers must exert additional effort to adapt exploits to each unique variant. Such work increases the cost of cyber attack and increases the probability of detection, serving as a deterrence to attackers.

A recent survey paper by Larsen et al. [25] highlights a number of issues related to the efficacy of software diversification techniques. First, many security evaluations are qualitative and based on logical argument. Existing quantitative metrics, such as entropy, have not been shown to be correlated with the cost to an adversary. Larsen et al. specifically call for the creation of methodologies to evaluate the impact of diversity techniques. Our work is an initial step in this direction, as we explore a quantitative way to measure the variation in the building blocks used by attackers to construct code reuse attacks.

A gadget is the building block of a ROP attack, a class of code reuse attacks designed to circumvent executable space protection [33]. Executable space protection allows a memory page to be either writable or executable but not both at the same time. Successful reuse of an exploit against diversified application variants requires all variants to share the same code snippets used in the ROP attack. Therefore, we provide a quantitative analysis of the effect of diversity techniques on the percentage of ROP gadgets common to multiple variants.

Our analysis focuses on the use of a diversifying compiler to automatically generate unique, but functionally identical, versions of an application. Unlike an ordinary compiler that is deterministic, a diversifying compiler is not expected to produce the same output when presented identical input [15]. For example, it need not eliminate dead code or, more precisely, probabilistically decides when to remove dead code. In addition, a diversifying compiler is free to introduce non-functional code such as NOPs and explore various orderings of instructions and basic blocks [14]. Different register allocation strategies, memory layouts, and optimizations (e.g., loop unrolling) all introduce variance in the executable [25].

Although one concern is that such variability in the output executable will not guarantee runtime or size optimality, a small loss in optimality is likely tolerable if the resulting variance significantly increases the cost to an attacker. Even optimizing compilers rarely have sufficient information to achieve optimality for all inputs. As performance is often the deciding factor when determining when to deploy new security techniques (adoption often requires the overhead to be less than 5–10% [38]), the performance overhead of software diversity techniques has been well-studied [25, Table III]. Hence, we focus on the security impact of software diversity techniques. Furthermore, diversity shows promise against classes of attacks (e.g., side channels) that are not addressed by traditional cyber defense techniques.

The contributions of this paper are as follows.

- We define a novel approach to measure ROP gadgets shared across a set of diversified application variants. Our bag of gadgets metric addresses diversity's impact even in the presence of attack techniques that dynamically discover available gadgets (e.g., JIT-ROP [37]).
- We use two quantitative approaches, Survivor [18] and bag of gadgets, to measure ROP gadgets shared across a set of diversified application variants. These approaches represent the amount of prior knowledge available to an attacker: no prior knowledge of the target instance of an application and an attacker who knows the existence of gadgets but not necessarily their location in the target instance.
- We apply diversification techniques to a commonly-used set of utilities (the GNU core utilities) to evaluate the impact of diversity to the ease of constructing a successful ROP attack.
- We show that the diversity techniques that we studied reduce the common set of gadgets by an order of magnitude, even for small population sizes. Nevertheless, the effectiveness of these techniques quickly plateaus, as some gadgets are common to all variants.

The remainder of this paper is organized as follows. Section 2 provides an overview of ROP attacks and address space layout randomization (ASLR) as a current defense against these attacks. Section 3 provides an overview of software diversity, specifically compiler-based diversity techniques. Section 4 explores the impact of these techniques to ROP gadgets. Section 5 discusses these results and their implications to the premise of software diversity. Section 6 contrasts our work with prior efforts in this area. We conclude and describe future work in Section 7.

2. RETURN-ORIENTED PROGRAMMING

A buffer-overflow exploit allows an attacker to redirect the flow of execution by overwriting a return pointer beyond the bounds of a buffer. In its most basic form, this exploit is the premise for a code-injection attack, wherein the attacker inserts malicious code into an executable and redirects the control flow so that the injected code is executed. In response to such attacks, executable space protection was introduced, whereby each page of memory is marked as either writable or executable, and any attempts to execute machine code on a page marked non-executable will result in an exception and, in most cases, immediate termination of the program. Thus, executable space protection makes it difficult for an attacker to execute injected code, inspiring the development of a new class of attacks, including return-oriented programming (ROP).

ROP attacks forego the idea of code injection entirely, choosing to focus on the reuse of existing machine code to achieve the desired result. The core component of a ROP attack is the "gadget," a sequence of instructions that occurs immediately before a return instruction. A ROP gadget is a sequence of bytes, terminated by a byte that represents a return, that can be interpreted as instructions and parameters (e.g., registers or immediate values). Unlike a code-injection attack, where the attacker utilizes their own injected exploit, ROP attacks compromise the application by executing a "chain" of gadgets built from program and library code. Because control can be redirected to any arbitrary byte in memory, ROP gadgets need not comprise instructions from the originally compiled code. Therefore, any pattern of bytes can be executed if it lies in executable memory and decodes to valid instructions [33]. Such occurrences are likely more frequent on ISAs with variable-length instructions and relaxed instruction alignment, such as x86.

There exist a number of tools able to extract useful ROP gadgets from target binaries. Some of these are capable of compiling the attacker's code directly into a chain of gadgets, eliminating the need for the attacker to manually search through a list of gadgets to assemble the payload. Examples of such tools include ROPgadget,[1] ROPC,[2] and Q [34]. Thus, the cost to the attacker to construct a ROP attack has decreased, heightening the need for effective defensive measures.

Address space layout randomization (ASLR) has arisen in recent years as a potential solution to ROP attacks. The basic function of ASLR is to randomize the locations in memory at which each binary is loaded. As the use of gadgets requires knowledge of specific memory addresses, the randomization of memory layout was believed to be a reasonable means of inhibiting the construction of ROP chains. Unfortunately, ASLR is traditionally coarse-grained, meaning that the base address of the executable, stack, heap, and libraries are randomized but the leak of a single memory address compromises its security because objects appear at a fixed offset from the base address. Attackers may rely on invariants (e.g., fixed offsets for objects from the base address) to minimize the effort required to compromise multiple instances from a single exploit.

Recently, the effectiveness of ASLR against ROP attacks has come into question, with the emergence of just-in-time code reuse demonstrating that, given the occurrence of a memory disclosure vulnerability, it is possible to dynamically construct ROP chains for a program with ASLR [12, 33, 37]. Snow et al. [37] demonstrate that even fine-grained ASLR that randomizes individual objects independently (e.g., memory addresses and code) cannot prevent code reuse attacks in the absence of memory safety. Techniques such as Execute-no-Read (XnR) [3] or ISA randomization [4, 21] that negate an attacker's ability to read arbitrary memory are currently the best defenses against this type of attack. We explore the implications of just-in-time code reuse to software diversity in Section 5.

3. SOFTWARE DIVERSIFICATION

The concepts underlying software diversity apply to the entire software development life cycle (Figure 1). For example, safety-critical systems may use design diversity and redundancy to achieve reliability. Such techniques are too expensive for many applications because they require multiple implementations from a common specification [1]. Moreover, care must be taken to ensure that faults occur independently because the number of coincident failures has been shown to be higher than expected from an assumption of independence [23]. At the source code level, different language constructs can be used to diversify aspects of an implementation. For example, the selection of a particular execution path can be implemented via a multiway branch (e.g., `switch` statement) or series of conditional expressions (e.g., `if...else` statements). Template metaprogramming can obfuscate applications automatically [29]. Compilers that intentionally modify their object code so each executable is unique are known as diversifying compilers [15]. Targeting a machine language unique to a particular execution environment is known as ISA randomization [4, 21].

This paper focuses upon diversification techniques that can be applied during compilation and linking. Targeting this phase of the application life cycle has a number of advantages, including reuse; existing compiler support for optimization, which minimizes the performance impact of

[1]https://github.com/JonathanSalwan/ROPgadget
[2]https://github.com/pakt/ropc

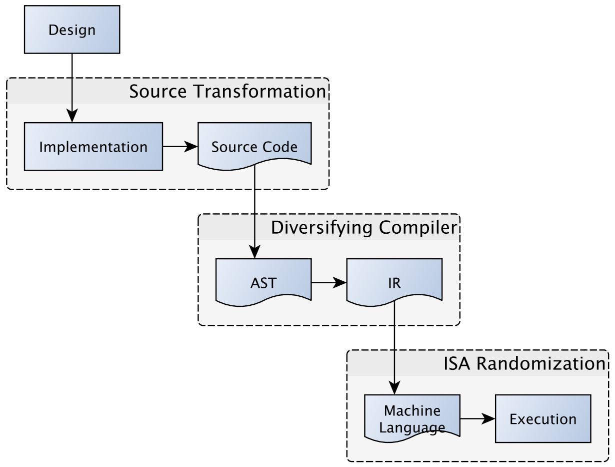

Figure 1: Opportunities for software diversity at various stages of the software development life cycle. The abstract syntax tree (AST) and intermediate representation (IR) are data structures used in the compiler.

some diversification techniques and aids the implementation of new diversity strategies; and whole program diversification. Applying diversification to other phases of the software life cycle has other advantages and disadvantages that are discussed elsewhere [24].

Compiler- and linker-based diversity techniques permit a broad number of different transformations, ranging from the instruction to program level. A brief description of the granularity of these classes of transformations and concrete examples of each are as follows. For more details, we refer the reader to Larsen et al.'s taxonomy [25].

Instruction These transformations operate within a basic block, a sequence of instructions with a single entry and exit. Changing or permuting instructions breaks fine-grained code reuse attacks, assuming that adversaries lack knowledge of the implementation details (e.g., via a memory disclosure vulnerability). Example transformations at this level include the substitution of equivalent instructions, reordering instructions, register allocation randomization, and garbage code insertion. Substituting equivalent instruction sequences and garbage code insertion theoretically allow an infinite number of variants although both may negatively impact the application's execution time and memory footprint.

Basic Block The order of basic blocks within a function may be permuted freely in conjunction with updating the addresses of branches and jumps. While there is a lower bound on the number of basic blocks within a function, basic blocks may be split to create additional opportunities for reordering. Branch functions and the insertion of opaque predicates that contain values known to the compiler but are difficult to deduce statically obfuscate the call graph and also hinder reverse engineering.

Loop Loops may be unrolled, partially or fully. When combined with the prior transformations, unrolled loop iterations may obfuscate the underlying computation

Data set	Version	Binaries	KLOC	Inst. (k)
GNU coreutils	8.25	103	60.1	700.0

Table 1: Statistics for the evaluation data set. The final column, "Inst. (k)," indicates thousands of instructions in the binaries.

(particularly in combination with other diversity techniques) and hinder side channel attacks.

Function These transformations vary the number and invocation conventions of functions. Specific techniques include stack layout randomization, altering the number and order of parameters, inlining and outlining, and control flow flattening.

Program These transformations randomize aspects of the entire program, including reordering functions in the executable, instruction set randomization [4, 21], data randomization, and randomizing library entry points.

Existing systems that implement some of these transformations are the multicompiler [18] and Obfuscator-LLVM [20]. Both systems are based on LLVM [27]. We use the transformations provided by the multicompiler and Obfuscator-LLVM as the basis for our study, as both are open source projects that are publicly available.

4. EVALUATING DIVERSIFIED POPULATIONS

This section describes our evaluation of the effectiveness of various diversity techniques. We start with an overview of our data set; describe the specific diversity techniques used in our study, our approaches to gadget counting, and experimental setup; and present our analysis of gadget survival for various diversity techniques.

4.1 Data Sets

We selected a single large data set for evaluation: the GNU core utilities.[3] Table 1 lists basic statistics about the data set. Thousands of lines of code (KLOC) was computed using David A. Wheeler's "SLOCCount" (version 2.26).[4]

The GNU core utilities is a set of common tools found on Unix-like operating systems. We selected the GNU core utilities as our data set for a variety of reasons. First, the source code is freely available, which is essential when evaluating compiler-based diversity techniques. Second, open source software supports reproducibility of our research. Third, the GNU core utilities comprises a large number of tools (i.e., applications). We use the variety of applications to investigate different trends across the collection, noting that individual applications provide only a single data point in our analysis.

For each diversification technique, we compiled 100 diversified variants of the binaries in the data set. Each diversified variant was compiled using a different random seed, targeting x86-64, and was dynmically linked. To compare diversified variants, Floyd's sampling algorithm [5] was used to select 4000 unique combinations of variants from the $\binom{100}{k}$ possibilities. This approach provides a uniform number of samples (i.e., an even random sample) for our analysis even though the total number of possible combinations varies significantly with the number of binaries being compared. The upper bound of 4000 samples maximizes the number of samples given the 100 available variants for each binary.

When applicable, our baseline for comparison is the same binary compiled by the multicompiler but with all diversification techniques disabled (i.e., LLVM 3.5).

4.2 Diversity Techniques

More details about the specific diversity techniques used in our evaluation appear in the following descriptions.

NOP insertion A NOP (the mnemonic for "no operation") is an instruction that the processor fetches and executes without modifying the register file or memory. The NOP insertion strategy [18] probabilistically inserts 0 or 1 NOPs prior to the current instruction. The type of NOP inserted may vary from the x86 instruction with the NOP mnemonic (i.e., `nop`) to instructions that preserve the processor state (e.g., `mov esp, esp`). In our experiments, there is a 50% probability of NOP insertion.

Instruction substitution In many cases, arithmetic operations can be expressed via different operations. For example,

$$b + c = b - (-c) = -(-b + (-c))$$

When possible, an equivalent instruction sequence is substituted for binary and boolean operations on integers. A complete list of the available substitutions appears in the Obfuscator-LLVM documentation.

Schedule randomization This transformation randomizes the instruction schedule. Conceptually, dependencies among instructions form a directed acyclic graph (DAG). With instruction schedule randomization, an arbitrary instruction is selected from those that are eligible to appear next, and the order of the remaining instructions is updated accordingly. In our experiments, 50% of the instruction schedule is randomized.

Bogus control flow This transformation modifies the control flow graph (CFG) of a function by adding a basic block prior to the current basic block. The new basic block contains an opaque predicate [10] that is difficult to deduce statically and guards a jump to the original basic block. For our experiments, bogus control flow is applied to all functions but the probability of inserting an opaque predicate is 30%.

Control flow flattening This transformation obscures the call graph by replacing direct jumps between basic blocks with indirect jumps through "jump tables" [26]. This obfuscation conceals the original structure of the program because all basic blocks appear at the same level and the execution path is controlled by a single variable.

Function shuffling This transformation permutes the order of functions in the object code generated by the compiler. Although global permutation across the entire executable is possible, our experiments use function shuffling on a per-object file basis.

4.3 Approaches to Gadget Counting

A ROP gadget is a sequence of bytes in the program that can be interpreted as valid, unprivileged, non-branching instructions that terminates with a return instruction. A challenge in our evaluation is how to compare ROP gadgets

[3]http://www.gnu.org/s/coreutils

[4]http://www.dwheeler.com/sloccount/

across a population of variants. We consider two different approaches to determine gadget identity. Identical gadgets in multiple binaries are valuable for attackers because these gadgets are invariant, having survived diversification.

Survivor [18] This metric considers a gadget's sequence of bytes and its program offset as part of the comparison. This metric assumes that a gadget is only useful to an attacker when the gadget has the same functionality at the same address. Otherwise the attack must be modified for use against other binaries.

Bag of gadgets This metric only considers a gadget's particular sequence of bytes in regard to uniqueness. If the same gadget appears in two different binaries, even at two different memory locations, it is considered a surviving gadget, available for use by an attacker.

Because this metric does not consider memory locations, it represents the maximum set of gadgets available to an attacker with prior knowledge about an instance of an application. Attacks that require a particular type of ROP gadget or a number of ROP gadgets of the same type will be foiled if these gadgets do not exist in the binary.

When comparing gadgets, we ignore intervening bytes that are NOPs, as NOPs do not affect the execution of the gadget.

For both metrics, the intersection of gadgets between all possible diversifications of a particular program or library are the core surviving gadgets, available to an attacker despite diversification. When identifying gadgets in a binary, we also only consider the binary's executable segments that are loaded into memory. Unless otherwise noted, we use the survivor metric as our primary evaluation metric.

4.4 Experimental Setup

We statically identify gadgets in Executable and Linkable Format (ELF) binaries by sliding a window across the binary's executable memory and disassembling the bytes within that window. This approach to identifying gadgets is similar to the Galileo algorithm [35]. Our approach handles misaligned parses of the original binary code snippet (such misaligned parses can serve as optimizations for a ROP attack [6]), and the counts of gadgets in the binaries include those resulting from misaligned parses.

The choice of the window size affects the number of ROP gadgets that will be identified. To determine an appropriate window size, we identified the total number of ROP gadgets across a variety of window sizes. Figure 2 shows the percentage of total ROP gadgets identified for each window size, assuming that a window size of 50 bytes is essentially an exhaustive search. As evidenced by the figure, the greatest density of gadgets appears within a relatively small window (2–11 bytes), but there is a long tail of ROP gadgets found for increasing window sizes. This tail is largely the result of including additional instructions with an existing ROP gadget. In practice, the number of side effects (e.g., register clobbers) increases with more instructions, making very long gadgets less likely to be included in a ROP chain.

To be conservative, we selected a window size that identifies 90% of all gadgets found—25 bytes for the GNU core utilities. This sliding window is larger than the default for other gadget scanners (e.g., ROPgadget and ROPC) so we feel that our choice for this parameter is conservative. However, larger gadgets are more likely to be "broken" by changes in the binary; the implications of this statement are discussed in Section 5. In addition, prior research assumes a maximum gadget length of 5 instructions [31] or notes that most gadgets can be constructed from a comparable number of instructions [8]. With a mean number of bytes per instruction of 3.85 and a median value of 3 (Figure 3), our window size significantly exceeds this search depth.

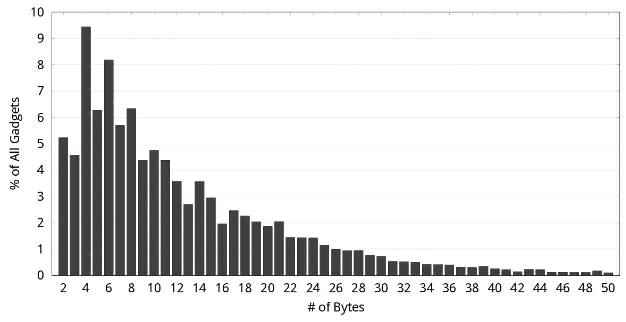

Figure 2: Percentage of ROP gadgets for increasing window sizes on the GNU core utilities.

4.5 Gadget Survival With Diversification

4.5.1 *Number of Gadgets*

In some cases, diversity techniques may actually increase the number of gadgets in a binary [37]. All the diversification techniques studied in this work were found to increase the overall number of unique gadgets across the population of diversified variants. As shown in Figure 4, the mean increase in gadgets ranged from 1% for function shuffling to more than 100% for control flow flattening. In the case of schedule randomization and instruction substitution, the new gadgets are primarily the result of a greater variety of instructions seen immediately before the function return. The situation is similar, if not immediately obvious, for control flow flattening where the order of the basic blocks can introduce new gadgets prior to the function return.

In addition to the change in the number of gadgets, the spread of the distribution indicates how uniformly the diversity technique applies to different binaries. Small spreads (e.g., function shuffling) indicate little variation, which is beneficial for our security analysis and obscures the presence of a diversity scheme to an attacker. Larger spreads indicate greater variability and may present a greater challenge to attackers due to the dissimilarity among variants.

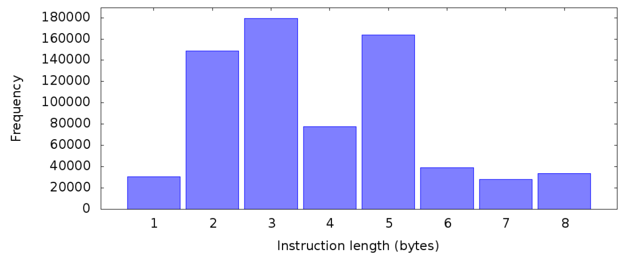

Figure 3: Histogram of bytes per instruction for the GNU core utilities.

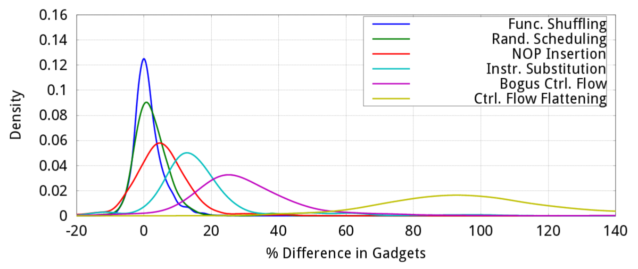

Figure 4: Change in number of gadgets across all GNU core utilities. Each line is a kernel density estimate of the probability density function; the bandwidth of the kernel is estimated using Silverman's rule of thumb [36]. In many cases, the distributions are close to normal distributions.

4.5.2 *Gadget Survival*

A more useful measure than the change in the number of gadgets is the effectiveness of each strategy at minimizing the surviving gadget set. We hypothesize that an attacker's effort increases as the common set of surviving gadgets decreases. Ideally, two diversified binaries should have absolutely no gadgets in common, and an attacker would have to build a new ROP chain for each diversified binary.

Figure 5 displays the percentage of surviving ROP gadgets across our even random sample of variant combinations with sizes from 2 to 16 binaries using the survivor algorithm to count gadgets. For each diversity technique, Figure 5 graphs the mean of the median values of surviving gadgets for each binary in the GNU core utilities. Of the strategies evaluated, we found control flow flattening to be most effective. Comparing two binaries, 1% survived on average, which is the core surviving set of gadgets. All diversification techniques except function shuffling eventually achieve a survival rate of less than 5% for larger sets of binaries. Function shuffling and schedule randomization improve rapidly from 2–4 binaries, starting at a 26% and 9% survival rate but eventually converging to a 8% and 3% survival rate.

The two primary characteristics of note in Figure 5 are the rates of decrease and the final percentage of surviving gadgets. Interestingly, most diversity techniques do not substantially increase their performance when considering larger sets of binaries. This result suggests that the output from most diversity techniques is uniformly distributed among their possible range of values; otherwise, one would expect the core surviving set to decrease with additional binaries. Also remember that each median point is selected from 4000 samples among the $\binom{100}{k}$ variants so the uniformity of the distribution may only be applicable when considering a larger number of combinations. Conversely, both schedule randomization and function shuffling require several binaries to maximize effectiveness. Initially, each retains a much higher percentage of surviving gadgets than the other techniques; however, schedule randomization improves rapidly and outperforms instruction substitution for groups with 4 or more binaries. We posit that the cause is a greater number of permutations in the instruction schedule than available arithmetic identities for integer and boolean operations; integer heavy benchmarks might have significantly different results than the GNU core utilities. Function shuffling converges to approximately 8% percent, a value over twice that of the other techniques. Thus, function shuffling is not as effective as the other techniques at decreasing the percentage of surviving gadgets.

Figure 6 displays the percentage of surviving ROP gadgets across our even random sample of variant combinations with sizes from 2 to 16 binaries using the bag of gadgets approach to count gadgets. Remember that this metric does not consider the gadget's location in the binary and represents the percentage of gadgets that can be found by an attacker when adapting an existing ROP attack to a variant. The difference in effectiveness as compared to the survivor metric (Figure 5) is striking. NOP insertion outperforms the other diversity techniques by a wide margin and eventually only 10% of gadgets remain common to multiple variants. Function shuffling performs very poorly, leaving more than 80% of gadgets common to all variants.

The bag of gadgets metric provides some unique insights into the effectiveness of the diversity techniques. First, even straightforward changes do have an effect on the ROP gadgets present across variants. An initial expectation that function shuffling would leave all gadgets present in all variants (albeit in different locations) is dispelled. Instead, we

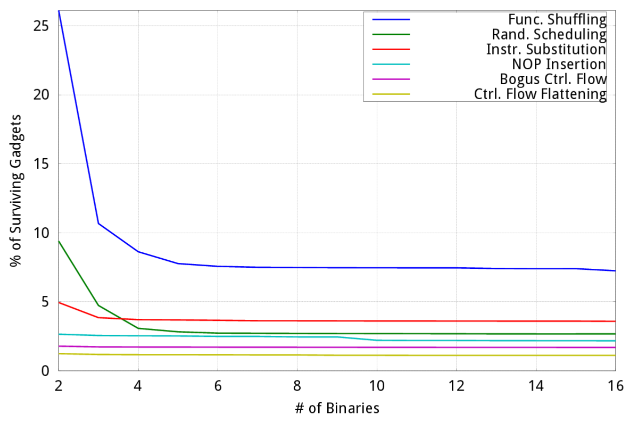

Figure 5: Median gadget survival across the GNU core utilities using the survivor metric.

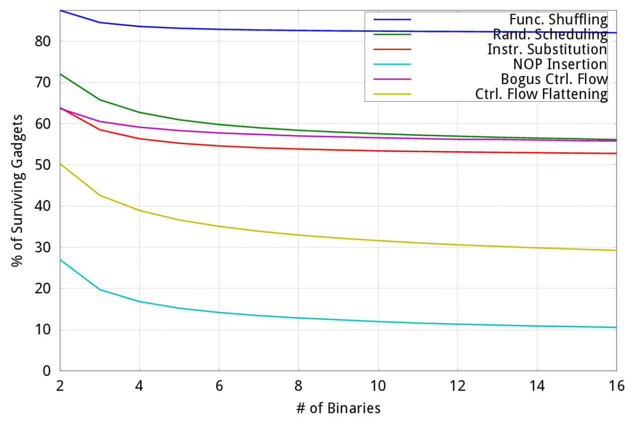

Figure 6: Median gadget survival across the GNU core utilities using the bag of gadgets metric.

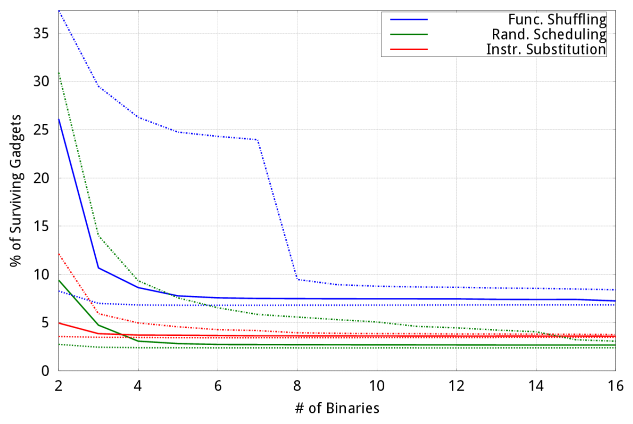

Figure 7: 5th and 95th percentiles (dotted lines) for gadget survival across all GNU core utilities using the survivor metric. The solid line is the median survival rate. Additional diversity techniques are omitted due to their similar performance to that shown in Figure 5.

see that some gadgets cross function boundaries, which accounts for the reduction in surviving gadgets. Techniques like NOP insertion may also prove reasonably effective at hindering current approaches to gadget scanning. Much of the reduction in the percentage of surviving gadgets for NOP insertion is due to the additional NOPs bloating the size of the gadget so it exceeds the search window size. While it would be simple to adapt existing techniques to handle NOP instructions, inserting a sequence of instructions that preserves the processor state would be more difficult to handle statically by a gadget scanner. Second, all techniques leave roughly an order of magnitude more gadgets in the binary than suggested by the survivor metric. These gadgets are present in all variants, but their location differs in each binary. Hence, they are available to an attacker that can read memory (e.g., via an information disclosure vulnerability). We discuss the implications of this result in Section 5.

Figure 7 shows the 5th and 95th percentiles for gadget survival rates for the GNU core utilities. This graph is useful to identify the range of the distribution and ensure that there are not significant outliers that perform significantly worse than suggested by the prior graph. The general trend is similar to the median across the data set, but in some cases, there is a large range in the effectiveness of the diversity techniques for different applications (e.g., function shuffling for 2–7 binaries). Interestingly, we see that the range between the 5th and 95th percentiles narrows with more binaries. This result suggests that larger populations have more consistency with regard to the effectiveness of a given diversity technique.

Figures 5 and 7 are both normalized to the number of gadgets originally available. An obvious question is how many actual gadgets survive. If the original binary contained only a few gadgets, then a 1% survival rate gives an attacker very few gadgets with which to craft an attack. Figure 8 shows a histogram of the number of gadgets in the original, undiversified binaries. As indicated by the figure, a 1% survival rate leaves an attacker with only a small set of gadgets at the same memory locations. Many attacks against common applications require only a handful of unique gadgets

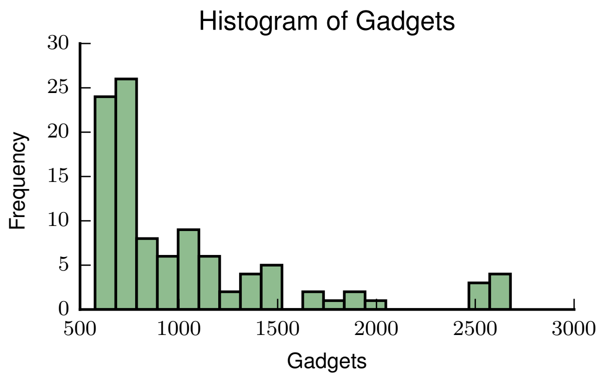

Figure 8: Histogram of the raw number of gadgets in the GNU core utilities. The data is collected from the undiversified binaries (median = 842 gadgets, $\mu = 1049.2$, $\sigma = 525.8$).

in the payload (approximately 10–20) [31]. Our analysis suggests that these diversity techniques are on the threshold of preventing these attacks outright due to the minimal number of surviving gadgets. An attacker is left with the challenge of adapting their payload to use these surviving gadgets, which may not always be feasible.

Furthering this initial analysis, we consider the number of opportunities for each diversification technique. Figure 9 considers function shuffling and shows the correlation between the number of functions and the number of surviving gadgets for the samples with two variants. A slight negative correlation exists between the values. A negative correlation is expected because more functions increase the number of permutations for function shuffling, which in turn decreases the likelihood that two variants will share the same permutation. In fact, for fewer than 100 functions, at least two variants must share the same initial function, but this overlap is not guaranteed when there are more than 100 functions in the binary. This trend suggests combining function shuffling with a complementary diversity technique like function outlining, which extracts new functions from portions of existing functions, because function outlining increases the diversification opportunities for function shuffling.

Function shuffling represents one extreme for our diversity techniques, as it is relatively coarse-grained, and the number of unique variants is limited to the number of permutations of the functions. At the other extreme is NOP insertion

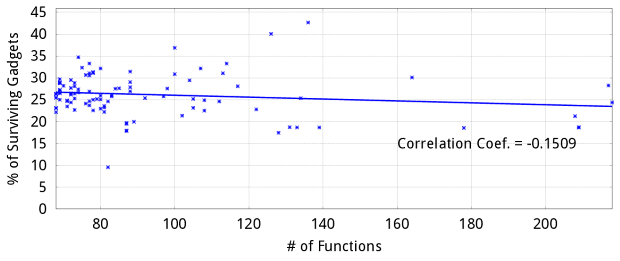

Figure 9: Correlation between the number of functions and surviving gadgets (survivor metric). The best fit line is the linear least squares regression.

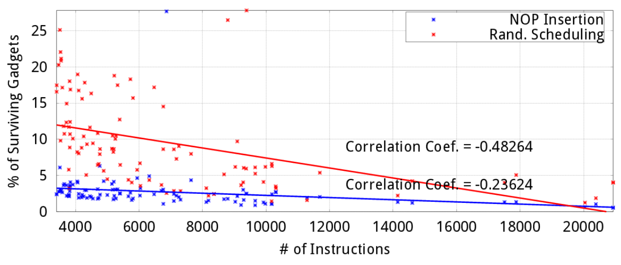

Figure 10: Correlation between the raw instruction count and surviving gadgets (survivor metric) for the GNU core utilities. The best fit line is the linear least squares regression.

because NOPs can be inserted between most instructions without modifying the program's functionality. Figure 10 shows the correlation between the raw instruction count in the original binary and the number of surviving gadgets. Again we witness a slight negative correlation. Even a single NOP can significantly reduce the surviving gadgets when inserted at the beginning of the text section although such a technique would be simple for an attacker to circumvent. Interestingly, the effectiveness of schedule randomization has a much stronger correlation with the raw instruction count. Further analysis is needed to determine the underlying cause, but an increased number of instructions allows more permutations in the instruction schedule in many cases.

We created a visualization to understand the relationship between the locations of gadgets common to multiple variants (Figure 11). In most cases, the surviving gadgets reside at the beginning of the binary where the diversity technique has had little opportunity to influence the output. On occasion, techniques like function shuffling will, by chance, align the same functions late in the binary, creating a zone of surviving gadgets deep in the binary.

This trend is clearly visible in Figure 12: surviving gadgets are fairly uniformly distributed in the presence of function shuffling. In comparison, gadgets that survive NOP insertion

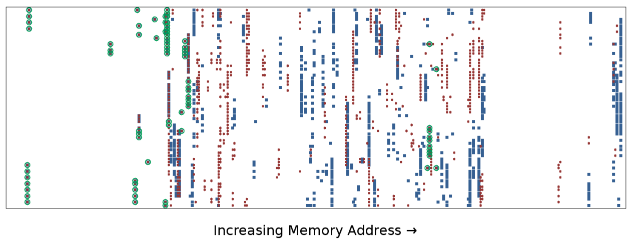

Figure 11: Gadget locations in two variants (red, blue) of `dirname` with surviving gadgets (survivor metric) circled in green. Function shuffling was used to diversify the two variants. This visualization shows the executable memory segments of the binaries normalized to the start of the first gadget (with some padding for visual clarity) and aligned on 64-byte addresses. Each column represents 64 bytes of memory (e.g., the first column shows memory addresses 0–63).

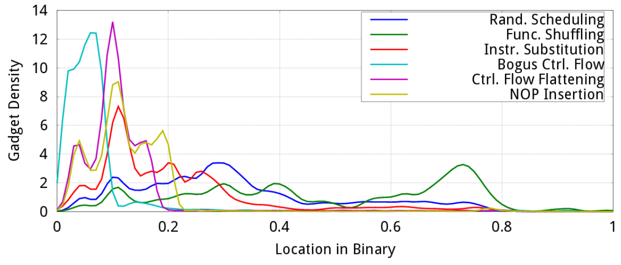

Figure 12: Location distribution of the gadgets that survive diversification (survivor metric). Each line is a kernel density estimate of the probability density function; the bandwidth of the kernel is estimated using Silverman's rule of thumb [36].

typically appear at the beginning of the binary when there has been fewer opportunities to alter the sequence of instructions. Techniques like bogus control flow that may insert a new basic block at the beginning of a function have an even greater impact on the location of gadgets, as the new basic block likely shifts the location of all later basic blocks and functions.

While it is logical to consider the combination of various diversity techniques, preliminary analysis with the survivor metric indicated that more effective techniques dominate other techniques, causing the combinations to perform similarly to the more effective technique on its own and not leading to substantial improvement. Hence, we do not consider the various combinations of these techniques here because the results are similar to those already presented.

5. DISCUSSION

Given a sufficient population size (e.g., more than 5 variants), all the diversity schemes reduce the common set of gadgets by at least an order of magnitude for the survivor metric (see Figure 5). However, the percentage decrease is dependent on the number of gadgets present in the variants. As previously stated, we chose a conservative search depth when identifying gadgets, but larger gadgets are more likely to be "broken" by diversification techniques, as there are typically more opportunities for diversification (at least for instruction-level techniques). To better understand the impact of the window size on our results, we compared the median gadget survival rate for the GNU core utilities with a sample size of 5 variants and window sizes of 10 and 25 bytes. Figure 13 compares the effectiveness; note that the values for 25 bytes as the same as that shown in Figure 5 for a sample size of 5 binaries. As evidenced by the figure, the change in the survival percentage is only minimally affected by the window size.

Unfortunately, the order of magnitude reduction that we previously reported in the number of surviving gadgets may not be sufficient to stop code reuse attacks. In many cases, the payloads of existing attacks only require a handful of gadgets—roughly comparable to the number that survive the diversity techniques that we studied. Nevertheless, it is encouraging that the effectiveness of some diversity techniques improves with larger binaries (see Figure 10). This improvement, though, does not translate into a reduction in the absolute number of surviving gadgets. Figure 14 shows

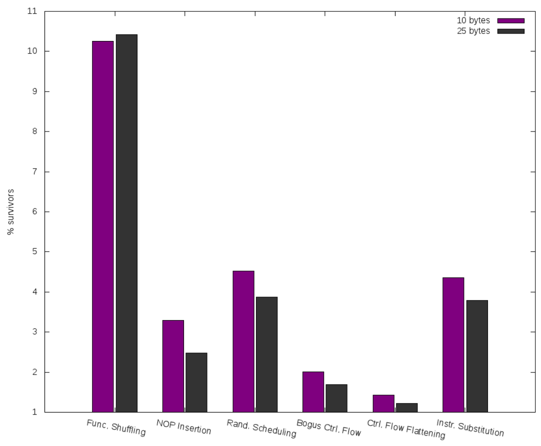

Figure 13: Comparison of the survival percentage for window sizes of 10 and 25 bytes (survivor metric) for samples of 5 binaries.

the correlation between the number of raw instructions and raw count of surviving gadgets, and it indicates that the number of surviving gadgets is essentially constant. Hence, the effectiveness of the diversity techniques that we studied appears to be on the threshold of preventing ROP attacks outright due to an attacker's inability to identify a set of gadgets that survive diversification. This conclusion changes if one considers the bag of gadgets metric, which shows a significant increase in the number of surviving gadgets as compared to the survivor metric. If an attacker can read arbitrary memory (e.g., via an information disclosure vulnerability), then it will be simple to adapt an existing attack to a new variant. Many gadgets are common across variants; the attacker need only discover their exact location.

Our analysis also suggests that a larger population size is unlikely to change this situation. The tail of our effectiveness distributions shows little change as the population increases from 6–16 variants (Figure 5). In combination with the raw number of surviving gadgets, this result casts doubt on the security promises of large-scale software diversity. Enough gadgets may survive diversification for an attacker to construct exploits from those gadgets. Other researchers' results (e.g., [31]) also suggest that some gadgets simply cannot be diversified away; our results support this finding.

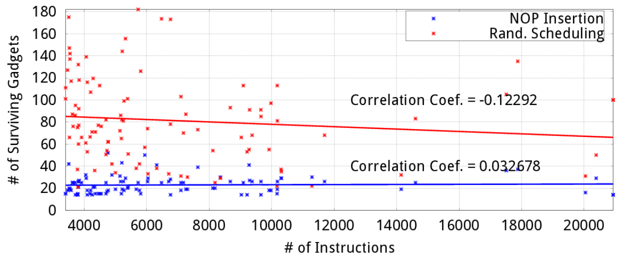

Figure 14: Correlation between the raw instruction count and raw count of surviving gadgets (survivor metric). The best fit line is the linear least squares regression.

Of course, attackers are still left with what might be a significant challenge. Is it possible to identify *a priori* those gadgets that survive diversification? If the core surviving gadgets are identifiable in some fashion, then attackers will naturally adjust their techniques to target the gadgets that are present in all variants. Although we did identify the likely location of surviving gadgets that appear at the same location in memory (see Figures 11 and 12), additional research is necessary to identify other relationships among these surviving gadgets that attackers will exploit.

Our approaches to gadget counting (see Section 4.3) do not allow for semantically equivalent gadgets. For example, a NOP may be represented as the NOP instruction or as an instruction that preserves processor state. In addition, instruction substitution may replace one arithmetic operation with a sequence of instructions that have the same side effects. Except for NOP instructions, we consider any changes to the instruction comprising the gadget to "break" that gadget, as it is no longer identical to the original. In the general case, semantic equivalence is undecidable, but this simplification potentially overstates the effectiveness of the diversity techniques, particularly for schedule randomization and instruction substitution.

Another open issue is just-in-time code reuse attacks. Snow et al. [37] demonstrate a technique that repeatedly abuses a memory disclosure vulnerability to map an application's memory at runtime, dynamically discovers available gadgets, and just-in-time compiles a target program as part of the exploit. Their technique is resilient to fine-grained diversity schemes including 1) function shuffling and basic block reordering, 2) instruction substitution and register randomization, and 3) schedule randomization. Moreover, they assume that these diversity schemes are performed on each execution of the application rather than being unique to each instance of the application. In the absence of complete memory safety, such attacks are difficult to prevent, but techniques such as XnR and ISA randomization may mitigate the immediate risks of memory disclosure vulnerabilities and complement the compile-time diversification techniques that we studied. In XnR [3], executable code pages cannot be read, thus preventing code reuse attacks. ISA randomization techniques that encrypt code pages [4, 21, 30, 32] do not prevent reading the code, but an attacker must break the encryption scheme before a code reuse attack becomes feasible. Given a sufficient key size and strong cipher, attackers are likely to look elsewhere for ways to compromise the application. It should also be noted that ISA randomization has a synergistic effect with compiler-based diversity schemes. Without the additional compile-time diversity, an attacker can easily construct a code reuse attack because all instances of the encrypted application are identical. That is, a successful attack against one instance protected by ISA randomization may be reused against other instances of the application unless those instances have additional differences beyond the encryption of code pages.

Our evaluation highlights some interesting opportunities to quantify the security of various diversity schemes. As shown in Figure 5, some diversity schemes are not affected by the size of the population whereas other diversity schemes increase in effectiveness as the population size increases. Techniques such as function shuffling are dependent on the number of functions in the application. Establishing strong theoretical bounds or even empirical bounds on the effectiveness of diversity techniques is an area for future work.

Finally, we do not consider the resistance of the various diversity techniques to reverse engineering. Some techniques like NOP insertion are likely trivial for an adversary to bypass. In the worst case for the attacker, the inserted NOPs change the location of the gadget in memory, but the (possibly bloated) gadget remains present in the binary. The same is generally true for function shuffling. The ideal diversification scheme satisfies three objectives:

- attack resistance,
- reducing exploit reuse among diversified variants, and
- resistance to reverse engineering for vulnerability detection and detection of the diversity details.

Our focus in this paper has been the second objective (i.e., reducing exploit reuse), but our existing metrics do not address the resistance of each technique to reverse engineering. We believe that good diversity schemes should follow Kerckhoffs' Principle [22]: i.e., the success of diversity should depend only on the secret of a random seed (e.g., of the diversifying compiler). Nevertheless, there is significant overlap between diversity and obfuscation techniques, and the latter may be beneficial to thwarting some attacks.

Threats to Validity

Our focus in this paper is specifically ROP gadgets, sequences of valid, non-branching instructions that terminate with a return instruction. However, ROP is not the only type of code reuse attack—e.g., Bletsch et al. [7] show how to maintain malicious control flow without return instructions. Our analysis does not consider additional types of gadgets. To do so, we would need to expand our search to include instruction sequences that end in a free branch (e.g., a return, jump, or indirect call). These additional gadgets may significantly alter our results, and it should also be acknowledged that the evidence of additional gadgets suggests strongly that an attacker has sufficient building blocks to construct an attack.

Finally, it should be noted that our results may not be indicative of diversifying compilers' success for larger, more complex applications. While some diversity techniques improve effectiveness for larger applications, the GNU core utilities are much smaller than other common applications such as office productivity suites or web browsers.

6. RELATED WORK

Larsen et al. [25] provide a systematic overview of software diversity, highlighting opportunities to diversify software and open challenges that exist in this field. In their words, "the study of how diversity affects the adversary's effort is in its infancy" [25]. Our work is one of only a few studies (e.g., [11,18]) that quantitatively evaluate the security impact of diversity strategies.

Early work on the multicompiler evaluated the percentage of gadgets that survive diversification. The Survivor algorithm [18] looks for shared instruction sequences that end with a free branch and appear at identical offsets within binaries. NOP instructions that may appear in these sequences are ignored, as they do not affect an attacker's ability to execute a code reuse attack. Our consideration of six diversity strategies significantly expands upon the prior results [18] that consider only the NOP insertion strategy. Like the prior results, we find that a subset of gadgets survives diversification of at least 12 variants, but unlike the prior results on the SPEC CPU benchmarks, we find that a much smaller set of binaries achieves maximum effectiveness.

Coppens et al. [11] examine matching the amount of code shared among diversified variants. This metric is a quantitative way to measure the effectiveness of diversity schemes, but unlike our work, it does not directly relate to specific classes of attacks (e.g., code reuse).

7. CONCLUSION

Our most striking conclusion is that the number of ROP gadgets that survive diversification, appearing at the same locations in multiple variants, is close to the threshold of that required to construct a ROP attack. We find this number essentially constant across the GNU core utilities, and our result is similar to prior results that suggest some gadgets simply cannot be diversified away using existing techniques. By itself, this result casts doubt on the effectiveness of stopping code reuse attacks solely through existing software diversity techniques. Moreover, if an adversary has an existing exploit for a particular variant, then our results also suggest that only minimal effort is required to adapt that exploit to another variant known to the attacker.

Nevertheless, these statements do not imply that software diversity is without merit. Attackers without access to a particular variant face a significant challenge of identifying the common set of gadgets that are always available for an attack. With only 1–10% of gadgets surviving diversification at the same memory location, this task may be expensive, error prone, and alert defenders of ongoing attacks via application crashes or other failures. While our preliminary investigation suggested little benefit in composing diversity techniques, that conclusion is likely specific to the diversity techniques—and possibly the data set—that we studied, as some diversity techniques described in the literature do have synergistic effects.

Future Work

Throughout the discussion of our results (Section 5), we identified several opportunities for future work. We summarize these opportunities briefly before highlighting other extensions to our work. First, additional research is needed to establish theoretical bounds on the effectiveness of diversity techniques. While we provide an empirical analysis in this paper, our results might not be representative of all applications. Second, the interaction between various diversity techniques when they are composed has received insufficient attention in the literature to date. Third, given that some gadgets appear to always survive diversification, is it possible to identify these gadgets without access to variants from the population as a whole? If an attacker can easily identify the core surviving gadgets, then they will simply target these gadgets to construct gadget chains, and diversity may not be a serious impediment.

Beyond the aforementioned questions raised by our analysis, there are a number of straightforward extensions to our work. First, we considered six different diversification strategies, but there are a variety of additional strategies that have been previously described in the literature. One technique, code layout randomization (i.e., basic block reordering), has been used by a variety of systems [11,13,40]. Including code layout randomization provides a common baseline between our analysis and prior work. Second, a sufficient variety of surviving gadgets are needed to construct a useful ROP chain. Our work does not account for the different kinds of of survivors, and whether or not enough flexibility survives to mount an attack. Third, we examine only one architecture (x86-64) and further analysis is needed to understand the degree to which the unique characteristics of x86 impact our results. For example, our analysis could be extended to other architectures, such as ARM, where instruction encoding and alignment is more rigid.

In addition to answering these questions, we envision the research benefits of developing a small library to automate much of the analysis performed in this paper. Such a library allows other researchers to perform a comprehensive analysis of additional diversification techniques and compare their effectiveness with others that have been previously studied. One potential drawback of the library is that it might help attackers circumvent a diversification scheme, but its research value appears clear.

Further, new metrics should be created to quantify the security impact of various techniques as a function of other statistics that are readily available (e.g., the number of instructions or functions in a binary). Significant work also remains to understand the composition of diversification techniques. For example, while function shuffling theoretically increases in effectiveness with the number of functions, function inlining decreases the number of functions, which eliminates opportunities for diversification. What combination of both techniques maximizes effectiveness? Despite the variety of diversity techniques that exist, many questions remain about their effectiveness and practical implementation; this paper is an early step in that direction.

8. REFERENCES

[1] A. Avižienis, M. R. Lyu, and W. Schutz. In Search of Effective Diversity: A Six-Language Study of Fault-Tolerant Flight Control Software. In *Proceedings of the 18th International Symposium on Fault-Tolerant Computing*, pages 15–22, June 1988.

[2] A. A. Avižienis. The Methodology of N-Version Programming. In M. R. Lyu, editor, *Software Fault Tolerance*, chapter 2, pages 23–46. John Wiley & Son Inc., 1995.

[3] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny. You Can Run but You Can't Read: Preventing Disclosure Exploits in Executable Code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1342–1353, 2014.

[4] E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, pages 281–289, New York, NY, USA, 2003.

[5] J. Bentley and B. Floyd. Programming Pearls: A Sample of Brilliance. *Communications of the ACM*, 30(9):754–757, September 1987.

[6] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh. Hacking Blind. In *2014 IEEE Symposium on Security and Privacy*, pages 227–242, May 2014.

[7] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: A new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 30–40, 2011.

[8] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented Programming Without Returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 559–572, 2010.

[9] F. B. Cohen. Operating system protection through program evolution. *Computers & Security*, 12(6):565–584, October 1993.

[10] C. Collberg, C. Thomborson, and D. Low. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 184–196, 1998.

[11] B. Coppens, B. De Sutter, and J. Maebe. Feedback-driven binary code diversification. *ACM Transactions on Architecture and Code Optimization*, 9(4):24:1–24:26, January 2013.

[12] L. Davi, C. Liebchen, A. Sadeghi, K. Z. Snow, and F. Monrose. Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming. In *22nd Annual Network and Distributed System Security Symposium*, NDSS 2015, February 2015.

[13] L. V. Davi, A. Dmitrienko, S. Nürnberger, and A.-R. Sadeghi. Gadge Me if You Can: Secure and Efficient Ad-hoc Instruction-level Randomization for x86 and ARM. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, pages 299–310, 2013.

[14] S. Forrest, A. Somayaji, and D. H. Ackley. Building Diverse Computer Systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems*, pages 67–72, May 1997.

[15] M. Franz. E Unibus Pluram: Massive-scale Software Diversity As a Defense Mechanism. In *Proceedings of the 2010 Workshop on New Security Paradigms*, NSPW '10, pages 7–16, September 2010.

[16] D. Geer, R. Bace, P. Gutmann, P. Metzger, C. P. Pfleeger, J. S. Quarterman, and B. Schneier. Cyber*In*security: The Cost of Monopoly. Technical report, Computer & Communications Industry Association, September 2003.

[17] J. R. Gosler and L. Von Thaer. Resilient Military Systems and the Advanced Cyber Threat. Technical report, Defense Science Board, January 2013.

[18] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz. Profile-guided Automated Software Diversity. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, pages 1–11, February 2013.

[19] T. J. John and R. Samuel. Herd immunity and herd effect: new insights and definitions. *European Journal of Epidemiology*, 16(7):601–606, July 2000.

[20] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin. Obfuscator-LLVM: Software Protection for the Masses. In *Proceedings of the 1st International Workshop on Software Protection*, SPRO '15, pages 3–9, May 2015.

[21] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering Code-Injection Attacks With Instruction-Set Randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, pages 272–280, October 2003.

[22] A. Kerckhoffs. La Cryptographie Militaire. *Journal des Sciences Militaires*, IX:5–38, January 1883.

[23] J. C. Knight and N. G. Leveson. An Experimental Evaluation of the Assumption of Independence in Multiversion Programming. *IEEE Transactions on Software Engineering*, SE-12(1):96–109, January 1986.

[24] P. Larsen, S. Brunthaler, and M. Franz. Security through Diversity: Are We There Yet? *IEEE Security & Privacy*, 12(2):28–35, March 2014.

[25] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. SoK: Automated Software Diversity. In *2014 IEEE Symposium on Security and Privacy*, pages 276–291, May 2014.

[26] T. László and Á. Kiss. Obfuscating C++ Programs via Control Flow Flattening. *Annales Universitatis Scientarum*

Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica, 30:3–19, 2009.

[27] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization*, CGO 2004, pages 75–86, March 2004.

[28] J. McHugh. Software Diversity: Use of Diversity As a Defense Mechanism. In *Proceedings of the 2005 Workshop on New Security Paradigms*, NSPW '05, pages 19–20, September 2005.

[29] S. Neves and F. Araujo. Binary code obfuscation through C++ template metaprogramming. In A. Lopes and J. O. Pereira, editors, *INForum 2012*, pages 28–40, September 2012.

[30] A. Papadogiannakis, L. Loutsis, V. Papaefstathiou, and S. Ioannidis. ASIST: Architectural Support for Instruction Set Randomization. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & Communications Security*, CCS '13, pages 981–992, 2013.

[31] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization. In *2012 IEEE Symposium on Security and Privacy*, pages 601–615, May 2012.

[32] G. Portokalidis and A. D. Keromytis. Fast and practical instruction-set randomization for commodity systems. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 41–48, 2010.

[33] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Transactions on Information and System Security*, 15(1):2:1–2:34, March 2012.

[34] E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: Exploit Hardening Made Easy. In *USENIX Security Symposium*, August 2011.

[35] H. Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 552–561, 2007.

[36] B. W. Silverman. *Density Estimation for Statistics and Data Analysis*, volume 26 of *Monographs on Statistics and Applied Probability*. Chapman and Hall, London, 1986.

[37] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *2013 IEEE Symposium on Security and Privacy*, pages 574–588, May 2013.

[38] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal War in Memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62, May 2013.

[39] C. Taylor and J. Alves-Foss. Diversity As a Computer Defense Mechanism. In *Proceedings of the 2005 Workshop on New Security Paradigms*, NSPW '05, pages 11–14, September 2005.

[40] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 157–168, 2012.