Überdiversity: Exploring the Limit of Load-Time Software Diversification

Thayer School of Engineering at Dartmouth College TR18-001

Scott Brookes, Martin Osterloh, Robert Denz, and Stephen Taylor

Abstract

Software diversification is a well-established method of mitigating the risk of vulnerabilities; ASLR functionality is included in most modern operating systems. However, most mainstream ASLR implementations do not use the maximum level of randomness available in hardware, and many fail to thoroughly discuss some of the "dark corners" of a diversity implementation. In this paper, we present a case study of our fine-grained load-time diversity implementation, "überdiversity"[1], in order to (1) examine the implications and feasibility of approaching the hardware-imposed upper bounds of ASLR and (2) formalize some of the frequently neglected details of diversity implementations. In particular, überdiversity makes several advancements over the state of the art including higher entropy, interleaving user- and kernel-space code sections in virtual memory, and randomizing the entire software stack from user process to hypervisor. We also motivate the need to discuss, explicitly, the algorithm used to produce random address space layouts and present an algorithm which does so provably uniformly at random. Additionally, we formalize the notion of *program variants* as a measurement that, along with the commonly used entropy measurement, can help to quantify the effectiveness of our and other randomization techniques. Although finding that the worst-case costs can be extreme, we measured a moderate runtime average performance impact of less than 10% using überdiversity, suggesting that mainstream ASLR implementations may be able to benefit from increased randomization without suffering undue performance costs.

Keywords

Diversity; Operating Systems; Hypervisor; Address Space Layout Randomization; ASLR; KASLR

1 Introduction

A significant concern in the modern computer security landscape is *vulnerability amplification*; since all network connected systems and programs are instances of a very small pool of possible operating system variants and programs, a single exploit can be reused on many machines. This includes classes of exploits such as buffer overflows [37], return-oriented programming (ROP) [49], `return-to-libc` [8], and many more.

Vulnerability amplification multiplies the potential reward for developing a single exploit many-fold. Injecting randomness into software systems is a very popular technique used to limit the impact of a vulnerability. In the case of load-time diversity, the attacker's workload is also increased dramatically because the effects of the randomization are not easily visible. Therefore, attackers are forced to design exploits without a priori knowledge of the layout of a system.

Load-time diversity involves loading some number of discrete units of code or data (sections) into memory at random locations. Überdiversity follows the example set by [56] by loading the sections at different places within the vast virtual memory space with blank space in between them. Code sections typically represent individual functions, but additional sections include global data, the heap[2], and the stack. Although überdiversity does not explicitly randomize *within* these sections, it is conceptually compatible with other projects that ran-

[1] The prototype discussed in this paper is available as open-source software at `https://github.com/SCSLaboratory/BearOS`.

[2] More precisely, a pre-set maximum size is set aside as a special section in which the contiguous heap will grow at run-time.

domize within functions [23, 46], the heap [43], and the stack [9, 10, 20]. The address space generated after randomization is extremely unlikely to share exploitable addresses with other program instances loaded from the same source code.

Section 2 acknowledges some of the many approaches that have been developed to increase diversity in computer systems and examines some of them to motivate the goals and contributions of this paper. We begin our discussion of überdiversity in Section 3 by presenting the algorithm we use to load a program such that all possible instantiations of the program are equally likely outputs. Though frequently overlooked, this is an important property that ensures that the benefits of a randomization implementation are fully realized. In Section 4 we present an overview of the implementation of our prototype in terms of the Executable and Linkable Format (ELF) Application-Binary Interface (ABI) and the x86 architecture. Section 5 estimates the effectiveness and measures the costs of our prototype.

The contributions of this paper include:

- A load-time diversity implementation that extends the state of the art by:
  - interleaving kernel- and user-space sections, producing an address space without unique regions designated for the kernel- and user-space.
  - simultaneously diversifying every layer of the software stack including the hypervisor, kernel, and user-space application.
  - providing a higher level of entropy that approaches the hardware-imposed maximum.
- Discussions of "dark corners" of diversity implementations that are rarely presented or formalized, including:
  - the properties of the algorithm used to generate a random program layout.
  - the number of program variants possible with an implementation: a measurement that compliments the traditional "entropy".

2 Related Work and Motivation

Computer system security via non-determinism was first proposed in various forms in seminal papers several decades ago [2, 11, 20, 48]. Since then, the idea has been fully embraced by the security community, leading to many research efforts exploring different applications and implementations of randomization for security at development-, compile-, load-, or run-time. Larsen et al. acknowledge the many works that came before überdiversity more thoroughly than the scope of this paper would allow in [33, 34].

A selection of the relevant works presented in [34] and some others is presented in Table 1. It summarizes which projects are 64-bit compatible, the maximum reported entropy that their implementation achieves, to what extent the publication discusses the algorithm used to implement their randomization, and which pieces of the software stack are being randomized: user (U), kernel (K), or hypervisor (H) code.

Table 1 summarizes several types of load-time randomization efforts. Some randomize the base addresses of entire program resources such as the stack, heap, code, data regions, or operating system modules [10,22,32,54]. [4] presents a comprehensive set of transformations to randomize these resources and introduce entropy within each such as randomizing stack variable layout and code function order. Other techniques use binary analysis to create programs with additional features: [55] produces code that randomizes itself when loaded, [23] produces code embedded with metadata that is interpreted by a virtualized run-time environment, and [32] permutes instructions and data in the binary. The code can also be shuffled after the primary loading and linking, but before the code begins execution [14]. [52] extends the other randomization techniques by choosing addresses that contain self-validating checksums that can provide some measure of run-time control flow integrity. Fine grained per-function load-time randomization is combined with compile-time transformations in [30].

The term "entropy" is often ambiguous and has had multiple interpretations in the literature. As an example, Davi et al. [14] claims that their implementation achieves $n!$ entropy, where n is the number of program resources being randomized. Although the claim that [14] can produce $n!$ address spaces is correct, the use of the term "entropy" in this context is fundamentally different from other works. Davi was actually discussing program variants, and this semantic confusion is not uncommon. *Both* program variants and entropy are important metrics for any diversity implementation. In this paper, we clarify the difference between these two complementary ideas and formalize a method for computing the less popular program variants measurement.

Although some of the other projects do use a 64-bit platform, the highest reported entropy is 29 bits in [32].

Study	64-bit	Maximum Reported Entropy (bits)	Discussion of Algorithm	(U)ser-space, (K)ernel, and/or (H)ypervisor
Chew & Song '02 [10]	✗	15	None	U & K
PaX '01 [54]	✗	24	None	U
Wartell et al. '12 [55]	✓	Unspecified	None	U
Shioji et al. '12 [52]	✗	Unspecified	None	U
Davi et al. '13 [14]	✓	Misspecified†	None	U
Bhatkar et al. '05 [4]	✓	26	None	U
Tanenbaum et al. '12 [22]	✗	Unspecified	Partial	K
Hiser et al. '12 [23]	✗	Unspecified	None	U
Kil et al. '06 [32]	✗	29	None	U & K
Kanter '13 [29]	✓	27	None	U & K
Überdiversity	✓	47	Full	U & K & H

† Davi's reported "entropy" is actually a count of program variants.

Table 1: Selected examples of fine-grained load-time memory randomization techniques.

With a close upper bound at 47 bits, überdiversity provides the highest entropy. This makes it particularly interesting as an exercise studying the effects of approaching hardware limitations, since commodity 64-bit hardware provides an address space of only 2^{48}.

None of the other studies surveyed included substantial discussions about the algorithm used to inject randomness. This is often overlooked as a trivial component of an implementation. In the case of simple ASLR, which might choose a single random address for a single loaded resource, this may be a fair assessment. However, several studies in Table 1 are performing more complex and fine-grained randomization. It is easy to consider an algorithm that seems to give random outputs but actually produces some address spaces with higher probabilities than others, and non-trivial to prove that an algorithm generates all possible address spaces with equal probability.

3 Diversification Algorithm

Algorithm 1 produces random program variants with linear running time. Due to these attractive properties, it is the algorithm used in the implementation presented in [29], although it is not discussed. However, it is an example of an algorithm that clearly does not produce address spaces with uniform probability. In it, the sequence of random numbers chosen is monotonically decreasing, therefore sections are more likely to have low addresses than high ones. An even more convincing algorithm, suggested as a uniformly random alternative to Algorithm 1 at an early stage of this research, is shown in Algorithm 2. This algorithm picks a list of random numbers, each one to represent a gap between two sections loaded into memory (or between a section and the boundaries of the virtual address space). Then, the numbers are scaled to fit the virtual address space. Surprisingly, even this algorithm does not produce address spaces with uniform probability.

Algorithm 1: Greedy Diversity Loader Algorithm. This algorithm produces address spaces in which any particular section is more likely to have a low address than a high one.

Input: The Address Space A & Sections to be Loaded S
Output: Random location s_i.location $\forall s \in S$

```
1 Randomly shuffle S;
2 for i = 1 to |S| do
3 |   Choose random k : 0 ≤ k < |A|;
4 |   s_i.location ← k;
5 |   A ← {0, 1, ..., k − 1, k};
```

This is because during the scaling step, random number sequences that are too small to fill the address space (need to be "scaled up") will have an effective minimum gap size between sections. In other words, program variants with a gap of size 1 between two sections are less likely than program variants with larger gaps, because they can be produced only by random number sequences that exactly match or exceed the size of the available virtual memory space. A formal proof of this intuition is found the Appendix.

While Algorithm 2 may be "random enough" for most

Algorithm 2: Scaling Diversity Loader Algorithm. This algorithm chooses enough random numbers for each to be a gap between two sections, then scales the random numbers to fit the available address space.

Input: The Address Space A & Sections to be Loaded S
Output: Random location s_i.location $\forall s \in S$

1 Randomly shuffle S;
2 Choose $|S|+1$ random integers from 0 to RAND_MAX $\{r_1, r_2, ..., r_{|S|+1}\} = R$;
3 $T \leftarrow f(R)$ where $f(R) = \left\lfloor R \left(\frac{|A| - \sum_{s \in S} s.\text{size}}{\sum_{r \in R} r} \right) \right\rfloor$;
4 s_1.location $\leftarrow t_1$;
5 **for** $i = 2$ **to** $|S|$ **do**
6 $\quad s_i$.location $\leftarrow s_{i-1}$.location $+ t_i$;

Algorithm 3: Generic Diversity Loader Algorithm. This algorithm chooses random numbers within the space of available loadable addresses, then walks the address space (skipping over already loaded sections) to find the address corresponding to that random number in the current state of the address space.

Input: The Address Space A & Sections to be Loaded S
Output: Random location s_i.location $\forall s \in S$

1 **for** $i = 1$ **to** $|S|$ **do**
2 $\quad$ Find A_i, the set of addresses available to load s_i;
3 $\quad$ **if** $A_i = \varnothing$ **then**
4 $\quad\quad$ **error**;
5 $\quad$ Pick random $k : 0 \leq k < |A_i|$;
6 $\quad$ Find the k^{th} eligible address: $a_{i,k} \in A_i$;
7 $\quad s_i$.location $\leftarrow a_{i,k}$

applications, it is not uniformly random across all possible address spaces. The choice between algorithms that modulate efficiency, ease of implementation, and degree of randomness should be an important and clear part of the design of any randomization scheme.

To contrast non-uniformly random Algorithms 1 and 2, we present an algorithm based on the "Fisher-Yates Shuffle" [19] which shuffles an ordered list to produce a permutation uniformly at random. In Section 3.2 we transform our algorithm into an instance of the Fisher-Yates Shuffle, proving that it produces a given permutation of the address space uniformly at random; given some input (and assuming a sufficient source of randomness[3]) no output is more likely than any other. The algorithm represents a less efficient but more random point on the spectrum of techniques from which a system designer can choose, and is shown in Algorithm 3.

One of the most powerful features of this algorithm is its ability to load sections randomly within a discontiguous virtual memory range. This flexibility allows, for example, loading some program within the entire available virtual address space, trivially avoiding the hardware-unimplemented "noncanonical" addresses that split the virtual address space into higher and lower halves. It also allows a unique capability when diversifying programs within the conventional paradigm of user- and kernel-space sharing a virtual address space. Specifically, a user process can be loaded into all available virtual memory space *after* the kernel is loaded. This means that at runtime, there are no deterministic address ranges for kernel or user sections. This is a departure from previous implementations of load-time diversity in which the kernel was diversified in isolation within a predefined "kernel-space" virtual memory range, and user programs only within a predefined "user-space" virtual memory range. Interleaving kernel and user sections adds a significant level of complexity for an attacker to deal with when developing an exploit.

[3]Insufficient sources of randomness can lead to non-uniformly random address space layouts as in [39].

3.1 Run-time Complexity and Termination Analysis

The algorithm iterates over a fixed set S, and nothing within the loop could cause a condition where it does not exhaust the set. Therefore, it will terminate.

The algorithm iterates over S and twice over A within the main loop (at lines 2 and 6). If the address space is represented as regions of available memory instead of individual addresses, the number of regions will be proportional to $|S|$. Therefore, the run-time complexity is $O(|S|^2)$.

There is a way to formulate FYS (and therefore may be a way to formulate the algorithm presented here) in $O(N)$ time [15]. However, in this case $|N|$ is the size of the available virtual address space. For any program, $|N| >> |S|^2$. The algorithm proposed here also has a much smaller memory overhead. For these two reasons, the version we've presented is actually better than its in-place alternative.

The algorithm will terminate, but there is also a simple condition for the more interesting outcome: completion without error. Expression 1 describes the condition necessary for the algorithm to succeed without an error.

$$\forall\, s_i \in S, |A_i| \neq 0 \tag{1}$$

First, we observe that an address space is big enough for S if it is at least as big as $2 \times |S|$ copies of the largest section in S. This is stated as Lemma 1 and proven the Appendix .

Lemma 1. *Algorithm 3 will always succeed to load a set of sections S with size $S.size = \sum_{s \in S} s.size$ if, $|A| \geq 2|S| \times s_l.size$ where $\nexists s \in S : s.size > s_l.size$.*

We can use Lemma 1 to show that the algorithm will successfully load the kernel without error. $|A|$ is as big as 255 TB before anything is loaded. According to Lemma 1, this would allow as many as 127,500 1 GB sections to be loaded. Since it is common for the entire kernel to be loaded into a virtual memory region of just 1 GB, this is clearly sufficient to load even the largest monolithic kernel into a blank-slate virtual address space. No kernel section will be large enough to divide the address space such that there is no sufficiently large memory space for some section. However, leveraging the full strength of this algorithm involves loading a user-space application into the virtual memory gaps left between the sections of a diversified kernel. We will show that in the typical case, this will not pose any problem in the loading of a user process. Additionally, the condition shown in Expression 1 can be satisfied even in the most dramatic case with only a small consideration made at the load-time of the kernel.

The memory footprint of a monolithic kernel such as Linux is less than a few gigabytes. As a worst-case scenario, assume that the memory footprint of a kernel is many orders of magnitude larger: 20 TB. In another worst-case assumption, assume that there are 1,000,000 unique sections for the kernel[4].

Using these assumptions, there will be 235 TB of virtual address space available for the user process. The concern is that this address space will be segmented into unusable chunks. The 1,000,000 sections will force the address space to be segmented into a maximum of 1,000,001 pieces for the user-space load-time iteration of the diversity algorithm. These partitions will *average* about 235 MB. This means that there will be at least one partition whose size is at least 235 MB; big enough for the entirety of many user programs. However, it is highly unlikely that sections will be placed so uniformly as to have only partitions of 235 MB remaining. The probability is high that somewhere within the 1,000,001 available partitions, there is a single one whose size is big enough to hold the entire user program as described in Lemma 1.

One can *guarantee* that an arbitrarily large user program will fit. Only a small modification is needed to the initial diversification of the kernel to do so; add a section s_u to the set of kernel sections, S_k, before the diversity algorithm is invoked to load the kernel. Allow s_u to be randomly placed according to the algorithm, but do not map any physical memory into the virtual address space corresponding to s_u, meaning that this section for the kernel becomes a contiguous region of eligible virtual addresses for the user process. This guarantees that at load-time of a user process, $|A| \geq s_u$.size. By controlling s_u.size, one can satisfy Expression 1 for arbitrarily large user programs. Lemma 1 shows exactly how large s_u.size should be for a given program. Note that making this allowance does threaten the claim for uniformly distributed outcomes; in an exceptional case where the entire user program must be loaded into s_u, user sections will be more closely grouped than if this method is not used.

3.2 Proof of Uniformly Distributed Variants

The benefit of randomization is increasing the number of possible variants of a program to suppress vulnerability amplification and to increase the search space of an attacker hoping to brute-force the system. These benefits are reaped most completely if the search space is distributed uniformly across all possible variants. That is to say that no variant of the program should be any more likely than any other variant. We can show that Algorithm 3 produces a permutation of sections within the address space uniformly at random by showing that it is actually equivalent to the Fisher-Yates Shuffle (FYS), which itself produces permutations of an array uniformly at random [19].

Throughout this section a particular visualization of the task of diversifying the address space will be helpful. The diversification process is modeled as the task of generating a random permutation P of the set N, where N is the set of sections to load, S, *combined with* the blank units of memory, B, that will be present in the address space after all of S is loaded. In other words, $N = S \cup B$ where $|B| = |A| - \sum_{s \in S} s.\text{size}$. Each permutation of N represents a unique variant of the program loaded into

[4]Ubuntu's `exuberant-ctags` package counted less than 500,000 functions in the entire source directory for Ubuntu version 10.10 and Linux Kernel version 3.13.0. Note that this includes all architectures and options, not all of which will be included in any particular kernel build.

Algorithm 4: Fisher-Yates Shuffle Algorithm [19]

Input: Some set N
Output: P, a random permutation of N
while $|N| \neq 0$ **do**
    Pick a random number $k : 1 \leq k \leq |N| + 1$;
    $P_{i++} \leftarrow$ the k^{th} element of N;
    Remove the k^{th} element of N;

a virtual memory context.

The Fisher-Yates Shuffle is shown in Algorithm 4. Figure 1a gives a visualization of Algorithm 4 and reflects the composition of N as the sections to load combined with the blank spaces to build a representation of the entire address space.

Figure 1b shows a small modification to FYS. The algorithm that this represents will be denoted FYS'. FYS' is the first step towards showing that Algorithm 3 is equivalent to FYS. Lemma 2 states that FYS' is still uniformly random is simple; the simple proof of this is shown the Appendix.

Lemma 2. *The algorithm (FYS') that differs from FYS by iteratively choosing a "place in line" for each member in N instead of iteratively choosing a random member of N to place "next in line" produces variants uniformly at random.*

Note that with the input given to the FYS' algorithm, some of the steps that it took were unnecessary. Because each of the "empty space" inputs is identical, all possible placements of these units results in an identical address space if the placements of the first sections is fixed. Therefore, assuming that all slots are empty and then running the algorithm for just S produces an identical result as running the algorithm with N (S and the empty spaces).

This transformation arrives at Algorithm 3 while maintaining the FYS property of producing variants uniformly at random at each step. Therefore, Algorithm 3 produces variants uniformly at random.

Finally, note that running the algorithm twice (for kernel-space and then for user-space) maintains the uniformly at random property. The order of the input sections does not matter, and running the algorithm twice is really equivalent to just pausing to do some other work between loading the kernel sections and loading the user sections. However, no other work changes the conditions required for the algorithm. Since doing the algorithm twice can be reduced to doing it just once with different input, the sequence of doing it twice maintains the uniformly random properties that were proven for running the algorithm once.

4 Implementation

The überdiversity prototype was implemented on Bear [41], a Minix-like research microkernel with an integrated hypervisor that supports full 64-bit multicore Intel hardware. While fully featured, the kernel and hypervisor are less than 10,000 lines of code, with extensive sharing between the components.

It is beyond the scope of this paper to provide a detailed specification for the implementation of our prototype. Instead, we provide references to resources describing the different components of the implementation and add discussion of the details specific to our prototype.

4.1 ELF & the Diversity Loader

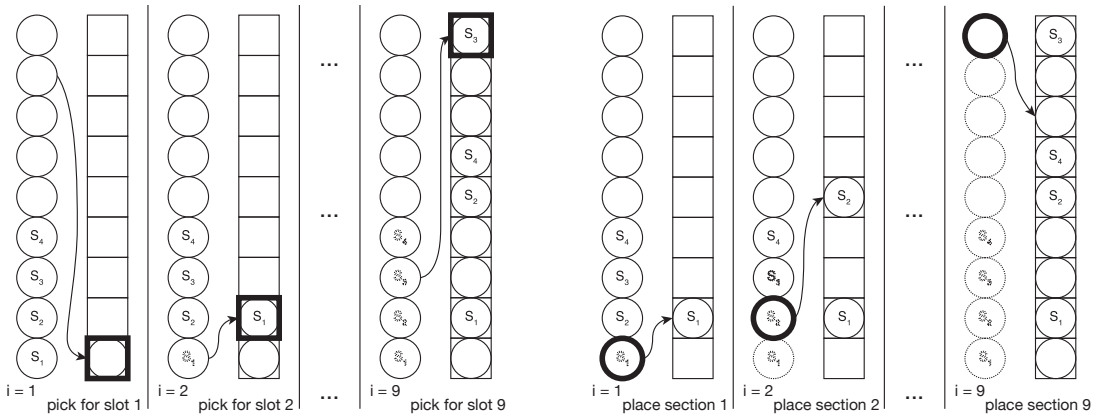
The compiler stores a program's code, data, and metadata in the ELF [12] format. In order to maximize the impact of load-time diversification, the compiler should take care to split the code into as many separate ELF sections as possible[5]. The ELF ABI and the associated loading process is well documented [36]. Adding support for diversification requires three additional steps. First, sections are loaded at randomly chosen addresses rather than those specified in the ELF file. After loading each section into a random virtual address, the diversity loader calls two routines per loadable unit: `move_unit` and `fixup_unit`.

The `move_unit` function updates the global symbol table loaded from the ELF metadata to reflect the new addresses of each section. It also updates internal relocation entries associated with the unit. These are places in a particular section that reference an absolute address within the same section.

After all units have had their symbols and relocations updated with their new randomized values, `fixup_unit` is called for each of the units. This routine resolves *external* relocations for the unit. In other words, after all of the symbols have been updated, each unit finds the updated value of the symbols it needs to resolve.

Random numbers are notoriously difficult for computers to generate. Sources of randomness are out of scope for this paper, but there are many published research efforts examining how to obtain random numbers from a

[5] In GCC, this is accomplished with the `-ffunction-sections` option



(a) Representation of the Fisher-Yates Shuffle

(b) FYS'; Modification of the Fisher-Yates Shuffle

Figure 1: The original Fisher-Yates Shuffle and an equivalent but slightly modified FYS'. Each produces permutations uniformly at random. FYS' is a step on the path to transform Algorithm 3 into FYS.

computer [47]. Our prototype uses the x86 hardware random number generator, accessed using the `rdrand` instruction. Despite some controversy about closed-source manufacturer provided hardware random number generators [42], this is sufficient for our proof of concept prototype.

4.2 The Virtual Memory Abstraction

The page tables [26] used by the MMU on the 64-bit x86 architecture are four levels deep. Each level contains protection bits used to moderate software's access to the memory mapped below that particular entry. These include a `U/S` bit to differentiate between user mode and supervisor mode access, a `R/W` bit to control read or write access, and an `NX` bit to control execution privilege, among others.

Despite the fact that these permissions are present at each level of the page tables, the protection bits at higher levels are not used in our prototype. An `NX` bit set in the top level table marks every virtual address decomposed through that particular entry as non-executable. With the maximum degree of randomization, however, there are likely to be multiple sections of different permissions interleaved within a single high-level table entry. In order to maximize the possible entropy achieved at load-time, permissions are enforced at the smallest possible granularity. For this reason, higher level table entries are always set with the most liberal permissions possible. The permissions required for a given section are set only in the final table entry. A small performance improvement is possible by marking higher-level tables that happen to contain only sections of a given type, but our prototype does not implement this case.

The entropy that can be achieved by the diversity loader has an upper bound defined by the amount of virtual memory space into which a given section can be loaded. In the case of the current x86 64-bit implementation, there are 48 bits of addressable virtual address space. This corresponds to the capability to virtually address 256 TB of memory. There is an architectural definition that, until the full 64-bit address width is implemented on the memory bus, the address must be sign extended. This creates a region of addresses that are illegal to access, called *noncanonical* addresses. The current 48-bit implementation of x86 does not allow addresses between `0x7FFFFFFFFFFF` and `0xFFFF800000000000`.

The fact that the virtual address space is so large allows for the large degree of entropy that can be achieved by harnessing the entire available virtual address space for loading sections. Recall that the flexibility of the algorithm described in Section 3 makes avoiding the noncanonical address range while still utilizing the entirety of the virtual address space trivial.

Interleaving kernel- and user-space sections raises concerns about the accessibility of kernel memory by unprivileged user processes. For instance, if the user stack is loaded directly adjacent to some kernel code, one might worry about a stack overflow attack to overwrite kernel code. The hardware memory management unit (MMU) mitigates this risk with hardware protections stored in the page tables. If the kernel code is loaded with the supervisor bit set in the page tables, any user process will fault trying to read or write that memory. Unfortunately, the page table protections are set and

enforced only in PAGE_SIZE[6] intervals. This has important implications for the implementation of the algorithm: no two sections with differing permissions can share a page. The stricter rule that no two sections of *any* type can share a page simplifies the implementation. This rule is enforced by making sure that addresses on any page that a section is loaded onto are marked as unusable during Algorithm 3. So, removing addresses from the pool of eligible addresses is done in PAGE_SIZE increments. Note that this rule does not mean that the address of a loaded section must be on a page boundary.

4.3 Diversifying the Entire Software Stack

The idea of diversity was first applied to user-space programs. Running on top of the operating system means that there is a powerful program (the kernel) used to load user-space code, with arbitrary complexity.

Kernel code randomization (KASLR) was the natural next step. Some KASLR implementations use the normal kernel boot loader to provide the randomization functions [16] while others use a hypervisor to load the kernel [29]. Our prototype uses the former method. A first-stage bootloader (boot1) manages the processor configuration needed to enter 64-bit mode at which point a second-stage bootloader (boot2) performs the randomized loading of the kernel.

A fully general diversity loader implementation in boot2 allows for the additional accomplishment of randomizing a hypervisor that is loaded with our bootloader. The hypervisor is diversified in its own address space. While the kernel- and user-spaces share an address space, the hypervisor is not mapped in to any set of page tables used in the kernel. It's own unique set of page tables makes it a completely separate unit from the kernel- and user-space. However, there are still huge security implications to having the hypervisor be loaded non-deterministically. As security technologies improve, attacks on systems are being launched at layers closer and closer to the hardware [18]. The conventional paradigm used to relate the hypervisor to the operating system suggests that hypervisors will soon be subjected to the attacks that motivate KASLR [50].

The result of this particular software decomposition and the flexibility of our diversity algorithm is a system in which every layer of the software is loaded randomly into the available virtual memory space. This includes the hypervisor, the operating system, and the user process. Furthermore, the user process and kernel that share a virtual address space are randomized together, rather than being randomized individually in disjoint virtual memory regions.

[6]The typical value of PAGE_SIZE on the x86 architecture is 4 KB

5 Evaluation and Analysis

5.1 Quantification of Diversity Achieved

There are two different ways to quantify the "amount of randomness" (more practically: "difficulty to brute-force") that a particular diversity implementation achieves. The first, entropy, is a familiar notion discussed in many previous papers about diversity [4,10,45,51,54]. The second, program variants, is a separate notion that is sometimes confused with entropy, as in [14]. We will clarify the difference between these two measurements and show how to count the number of program variants possible from a single source binary.

Entropy, the common way to measure a diversity implementation, quantifies the search space for a particular resource. If a particular function is loaded with n bits of entropy, then its address is one of 2^n choices. Commodity 64-bit hardware imposes a 48 bit limit on entropy since it only implements 2^{48} virtual addresses.

Program variants, on the other hand, describes how many unique layouts can be generated within an address space with a given program as input. A single program variant is defined by the union of the locations of *all* instructions and data loaded into memory.

In order to illustrate the difference between entropy and program variants, consider the effect of randomizing instructions *within* a function at compile-time. This transformation does not increase the entropy beyond simply loading the ordered function at a random location in memory, since the search space for an instruction is the same regardless of its location relative to its parent function. However, it does increase the number of program variants possible.

The practical question of which of these is a more important measurement for estimating the difficulty of brute-forcing a diversity implementation is a challenging one that depends on the goal of the attacker. If the attacker needs to find one single program resource (such as a particular dangerous function), entropy is the more important measure since it describes the amount of randomness in the address of that function. If the attacker needs to find a series of program resources (such as gadgets for a ROP payload), program variants is the more important measure since it helps describe how learning one location narrows the search space for other locations. Indeed, a complete model for estimating the probability of success of an attacker depends on these variables and

many more. However, even in the absence of a fully developed model, both program variants and entropy are important for a reasonable estimate of the potency of a given randomization technique.

5.1.1 Entropy

Entropy is a convenient way to measure the potency and flexibility of a diversification scheme, but it is important to recognize its limitations. A measure of entropy is not necessarily a good measure of security. As pointed out in [33], an interesting instruction loaded with infinite entropy into an infinite address space is still insecure if all other instructions are `nop`. In this example, executing the `nop` instruction at address 0 guarantees execution of the interesting instruction as the many `nop` instructions preceding the interesting instruction will be executed with no effect until the interesting function is found.

With the limitations of entropy as a quantification for diversity in mind, we will discuss the entropy of our prototype. Our prototype is implemented with sufficient flexibility to take advantage of the entirety of the x86 48-bit address space, with only some exceptions.

Some of the address space is unused at the top level. In particular, two entries are reserved in the top level of the page table structure. The 511^{th} entry is used for a self-referential pointer used to provide virtual mappings to paging structures [38]. The 0^{th} entry is unused out of convenience; the bootloader must be loaded into low memory so the low memory region is reserved to avoid conflicts with boot code. There are other complications to using low memory such as the `memmap` and `RAMDISK`, structures loaded into low memory by the bios during boot that are used throughout the lifetime of the kernel. However, most or all of this section could be reclaimed for use during loading with some care.

Additionally, the least significant address space bits are sometimes unavailable for loading sections. Each individual section in an ELF binary has an alignment requirement [12]. The alignment requirement is reported in the `sh_addralign` member of each ELF section header. Many section headers report an alignment requirement of 0 or 1, meaning that its section requires no special alignment: all of the least significant bits of the address can be loaded arbitrarily. However, some sections require 2, 4, 8 or greater byte-alignment. For any `sh_addralign` greater than 1, $log_2(\texttt{sh\_addralign})$ bits of the address must be held at zero in order to conform to the ELF standard. However, all remaining low-level bits can be randomized. Unlike some prior work, our prototype can load sections off of page boundaries.

The inability to use the 511^{th} and 0^{th} entries in the top level of the page table structures does not cut the available addresses in half, so it does not cost a full bit of entropy. However, it does cost some fraction of a bit. Therefore, worst-case upper and lower[7] bounds on überdiversity's load-time entropy are 47 and 43 bits, respectively.

5.1.2 Program Variants

One way to measure the diversity of a system is in terms of the number of *program variants* that can be generated from a given binary. The set of all possible program variants is the search space for an attacker that wants to learn the layout of the address space by brute-force guessing. The larger that search space is, the greater the workload of the attacker. Similar to entropy, this measurement does have limitations. Many unique instances of a program will have a particularly interesting resource located in the exact same location. It is an open research question how likely this is to assist the attacker in practice.

The question of how many program variants are possible is a combinatorics problem that can be visualized using the common "stars and bars" method [17].

First, assume that for a best-case scenario, all sections $s \in S$ require only 1-byte alignment. Note that $S = S_u \cup S_k$; the sections to be loaded are both the user- and kernel-space sections. Also assume that the hardware enforces protections at a single byte granularity. These assumptions provide the situation for an upper bound on the number of variants.

Since $\sum_{s \in S} s.\text{size}$ is known, the total amount of virtual memory that will be unused after the program is loaded is known. If the amount of virtual memory eligible to be used for loading is $|A|$, there will be $|A| - \sum_{s \in S} s.\text{size}$ unused bytes after all sections are loaded. Each of these unused bytes will lie in a region between two sections or a section and a virtual memory barrier. In other words, the sections can be considered dividers that break the unused bytes up into $|S|+1$ "bins" or "buckets". Each possible assignment of unused bytes to bins describes a possible program image.

Counting the ways to place unused bytes into $|S|+1$ bins is accomplished using the following visualization. Imagine each of the $|A| - \sum_{s \in S} s.\text{size}$ unused bytes is a "star". If $|S|$ stars are transformed into "bars" (sections, in this case) then the remaining $A - \sum_{s \in S} s.\text{size} - |S|$

[7] The lower bound is reported for a worst-case `sh_addralign` of 8 bytes. This is typical for the largest alignment requirement in an ELF binary.

stars are divided into $|S|+1$ buckets. However, for this system to represent the diversity loading problem, $A-\sum_{s\in S} s.\text{size}$ stars must be divided into the $|S|+1$ bins. Starting with $|S|+A-\sum_{s\in S} s.\text{size}$ stars and picking $|S|$ at random to represent sections instead of unused bytes accomplishes this. Choosing k objects from a set of n is an elementary combinatorial problem. Therefore, an upper bound on the number of variants that can be produced from a given program is shown in Equation 2.

$$\frac{\left(|S|+A-\sum\limits_{s\in S} s.\text{size}\right)!}{\left(A-\sum\limits_{s\in S} s.\text{size}\right)!} \tag{2}$$

Unfortunately, alignment requirements and coarsely grained MMU protection mechanisms mean that this upper bound is not reached. To capture a better idea of the actual number of variants possible, we will make some adjustments to more accurately represent the problem. First, the MMU provides memory protections only at the granularity of `PAGE_SIZE` chunks. So, the memory consumed by a section will be in units of page size. For the lower bound, a safe assumption is that each section consumes the maximum possible amount of memory. This is $\left\lceil \frac{s.\text{size}}{\texttt{PAGE\_SIZE}} \right\rceil + 1$. Adding the extra page is just in case the section is loaded across a page boundary. This lower bound is expressed in Equation 3.

$$\frac{\left(|S|+\frac{A}{\texttt{PAGE\_SIZE}}-\sum\limits_{s\in S}\left(\left\lceil \frac{s.\text{size}}{\texttt{PAGE\_SIZE}} \right\rceil+1\right)\right)!}{\left(\frac{A}{\texttt{PAGE\_SIZE}}-\sum\limits_{s\in S}\left(\left\lceil \frac{s.\text{size}}{\texttt{PAGE\_SIZE}} \right\rceil+1\right)\right)!} \tag{3}$$

The lower bound does not achieve the true value because it assumes that sections are eligible to be loaded only at page boundaries *and* that sections are loaded such that each will cross over a page boundary, consuming an extra page. This is a contradiction, but that is acceptable for a lower bound. A tighter lower bound examines where a section will be loaded within the memory it can consume. In other words, multiplying the original lower bound by the number of places *within* the memory footprint consumed by a section that the section can be loaded. This bound is shown in Equation 4.

$$\frac{\left(|S|+\frac{A}{\texttt{PAGE\_SIZE}}-\sum\limits_{s\in S}\left(\left\lceil \frac{s.\text{size}}{\texttt{PAGE\_SIZE}} \right\rceil+1\right)\right)!}{\left(\frac{A}{\texttt{PAGE\_SIZE}}-\sum\limits_{s\in S}\left(\left\lceil \frac{s.\text{size}}{\texttt{PAGE\_SIZE}} \right\rceil+1\right)\right)!} \times \prod_{s\in S} \frac{\left(\left(\left\lceil \frac{s.\text{size}}{\texttt{PAGE\_SIZE}} \right\rceil+1\right)\times \texttt{PAGE\_SIZE}\right)-s.\text{size}}{s.\text{alignment}} \tag{4}$$

Equation 4 is a close lower bound. There is one thing that keeps it from being an exact calculation: the equation assumes that *every* possible position to which a section could be loaded would consume the maximum possible amount of memory for that section. However, in truth only a small subsection of the eligible addresses cause a section to consume $\left\lceil \frac{s.\text{size}}{\texttt{PAGE\_SIZE}} \right\rceil + 1$ pages. Most consume only $\left\lceil \frac{s.\text{size}}{\texttt{PAGE\_SIZE}} \right\rceil$ pages. Finding an exact value by including this in the calculation is beyond the scope of this paper.

We calculated the upper and lower bounds for a typical program loaded using our prototype implementation. We used a shell as the example program to calculate sample values. On our system, $A \approx 255\text{TB}$, $|S_u| \approx 143$, and $|S_k| \approx 444$. Furthermore, $\sum_{s\in S_u} s.\text{size} \approx 800\text{MB}$[8] and $\sum_{s\in S_k} s.\text{size} \approx 80\text{MB}$. This gives an upper bound of $\approx 10^{8440}$ unique address space layouts for a typical program and a lower bound of $\approx 10^{8205}$.

These numbers do not include the number of variants produced by any compile-time randomization techniques that are used; including these effects effects will greatly increase the number of possible program variants. This shows that, at its limit, diversification of software can generate a virtual address layout search space that will overwhelm and defeat brute-force attacks.

5.2 Performance Cost

The performance cost of this method of diversity is spent partially at run- and partially at load-time. At load-time, there is added work from running the diversity algorithm. At run-time, the source is slightly more subtle. Consider a series of very small functions; without load-time diversity, these are likely to all be located on a single page of memory. This means that a single translation lookaside buffer (TLB) entry or cache page will cache all of these

[8]The quantity $\sum_{s\in S_u} s.\text{size}$ is dominated by the user-space heap which is allocated with a size of several hundred MB in order to accommodate memory-intensive applications. Although the heap is allocated on demand as the user process requests it, a maximum size is assigned at load-time to guarantee that the heap has sufficient room to grow.

Fork/Exec Test		CPU Cycles$\times 10^9$
No Diversity	Max	5.2296
	Min	4.8050
	Avg	4.9768
Überdiversity	Max	36.4030
	Min	35.5999
	Avg	35.9565
Average Performance Cost		7.2×

Table 2: Fork/Exec Test - 50 Trials

functions. With load-time diversity, however, a TLB entry and cache page will be required for each of these functions. More importantly, there will be a cache miss the first time each of these functions is executed compared to the non-diversified case in which only the first time any of these functions is executed results in a cache miss.

The load-time costs of our prototype are summarized in Table 2. This table measures the time spent to create and transform a process using `fork` and `exec`. The average case did show a high cost of 7.2×. Fortunately, this cost is incurred only once in the lifetime of the process and therefore overall throughput is not heavily affected.

In contrast, run-time performance overhead can pose a real threat to overall system throughput. Caching plays a huge role in offering the performance necessary to make modern systems useable, and fine-grained randomization threatens that entire subsystem. Figure 2 explores a worst-case scenario. In particular, it shows how the performance cost increases with nested subroutine depth as the caching capabilities of the hardware begin to be overwhelmed by the number of entries required for effective caching. The tests used to generate the data for this figure simply called nested subroutines until the function at the desired depth returned a value. Performance costs as high as 400% are a testament to just how important caching is, and how it can be disrupted by fine-grained randomization.

Although the experimental status of the platform for the prototype limits the type of applications that can be measured, Table 3 illustrates measured performance costs on industry standard benchmarking utilities. The AIM9 Tests [1] demonstrate that, despite the high costs seen in Figure 2, code diversity has no meaningful performance cost on code execution within a function after load time. The `malloc` test [35] does show some performance due to the cache misses from `malloc` and all of its subroutines. Importantly, however, even though this test uses many subroutines in the C standard library, only a 6% performance cost was measured. This suggests

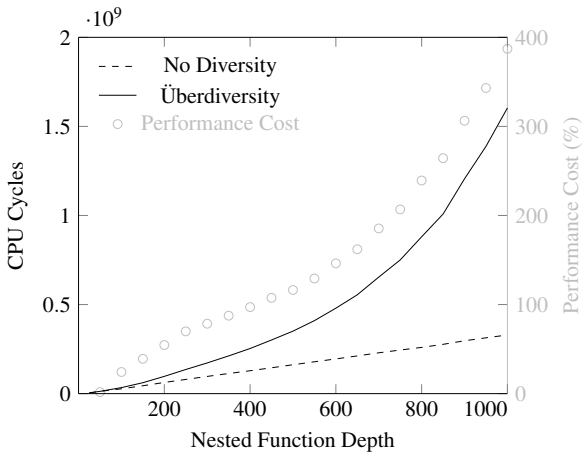


Figure 2: Performance Benchmarks for Nested Functions. This test illustrates that loading functions onto unique discontinuous pages instead of loading them contiguously can lead to cache exhaustion in extreme use cases.

that many applications follow the anecdote that "software spends 90% of its time in 10% of its code," and therefore that a very advanced level of randomization can be deployed with acceptable performance cost.

Finally, there is some memory overhead associated with using such a fine-grained randomization scheme. The code itself will have a larger memory footprint because memory is consumed in increments of `PAGE_SIZE`, but each function is loaded in isolation from the others. Therefore, two functions that fit together on one page will now consume 2 pages. Additionally, more memory is used to generate the higher-level paging structures that provide the mappings to randomized, non-localized addresses. Figure 3 demonstrates the worst-case of this cost is between 10× and 15×. This cost may be prohibitive for some applications or low-memory systems, but the amount of memory on state of the art systems is typically more than sufficient to accommodate this overhead for average workloads.

Furthermore, techniques used to decrease some of these performance and memory overheads are explored in [56]. This work explores optimizations such as bundling leaf functions with their callers to minimize an extra cache miss in a case where security is unlikely to be decreased. These same techniques could be applied in a 64-bit environment.

Test (CPU Cycles)		Multiply ($\times 10^{10}$)	Add ($\times 10^{10}$)	Divide ($\times 10^{11}$)	`malloc` ($\times 10^{11}$)
No Diversity	Max	2.2619	1.3811	1.0937	2.9638
	Min	2.2394	1.3512	1.0830	2.9563
	Avg	2.2432	1.3631	1.0851	2.9596
Überdiversity	Max	2.2647	1.8987	1.1037	4.2529
	Min	2.2388	1.3307	1.0373	3.0115
	Avg	2.2464	1.3674	1.0629	3.1393
Average Performance Cost		0.14%	0.32%	-2%	6%

Table 3: AIM9 Benchmark Suite [1] and Malloc Test [35] Performance Measurements - 50 Trials per test

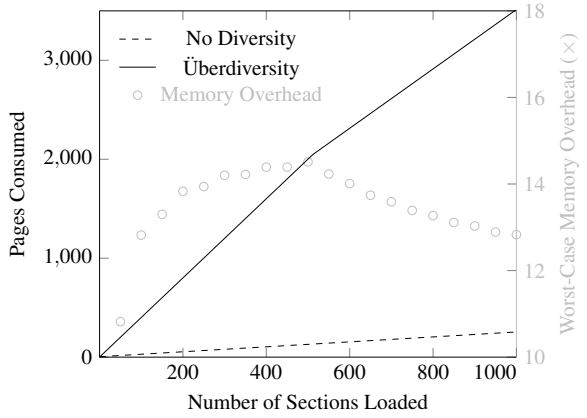


Figure 3: Worst-Case Memory Overhead Analysis. This analysis assumes that sections are $\frac{1}{4}$ the size of a page, and that the random locations are chosen in the most memory-heavy way possible.

5.3 Security Implications

As traditional ASLR has become ubiquitous in many commodity operating systems, attackers have developed a multitude of ways to bypass its protections. The various classical methods of exploiting programs despite ASLR are presented in the ASLR Smack and Laugh Reference (ASLRSLR) [40]. We will examine a selection of these methods and discuss how überdiversity influences their efficacy.

Aggression. The first and most obvious threat listed in ASLRSLR is a brute-force attack in which an attacker gains access to a vulnerable pointer that has been randomized by repeatedly guessing. For example, [51] shows that a system with only 24 bits of entropy can be brute-forced in a matter of minutes. Although [45] reported being able to brute-force a 64-bit ASLR implementation in as little as 1 hour, the ASLR implementation used was only 28 bits. As discussed in Section 5.1.1, our prototype achieves at least 43 bits of entropy. With 43 bits, a brute-force attack would take *months*[9]. This means that any attempted brute-force attack will be easily detected before it succeeds.

Return into Non-Randomized Memory Regions. In some ASLR implementations, there are some sections that are not randomized. Ways to exploit each of the text, heap, bss, and data sections when they are not randomized are detailed in ASLRSLR. Fortunately, our prototype successfully randomizes the location of *every* memory region. The ELF maintained relocation tables allow fixing pointers to data in the bss/data section and to functions to reflect the newly randomized locations. The user-space malloc implementation learns the location of the randomized heap by making a system call to request the address. Similarly, the kernel locates its heap at runtime by referencing data passed on by the bootloader.

Information Leakage Attack. Direct or indirect memory disclosures can defeat address space randomization by divulging address space layout information at run-time, enabling unique attack vectors [53]. Many methods such as execute-only code [3, 6, 13, 21] aim to limit the risk of memory disclosure. Most of these techniques are compatible with überdiversity.

Furthermore, fine-grained compile-time diversity should be used to reduce the amount of usable information that an attacker gains from any memory disclosure. Compile-time diversification should introduce randomness *within* an ELF section, denying the ability to make assumptions about the location of gadgets within a given ELF section by reading a single address off of the stack. Research efforts examining the necessary type of compile-time randomization are plentiful [24,27–29,31], and most are be compatible with load-time diversification techniques.

New information leakage techniques include side channel attacks, including methods developed specifi-

[9]Estimated. Experimental results from [51] and [45] say that 24 bits requires 4 minutes to brute-force, and 28 bits requires an hour. This conforms to intuition, as 4 min $\times 2^{(28-24)} = 64$min ≈ 1 hr. The same calculation for 43 bits gives 4 min $\times 2^{(43-24)} = 524288$ min ≈ 1 yr.

cally for defeating ASLR [25]. However, the authors of [25] state explicitly that "by utilizing the complete memory range and distributing all loaded modules to different places, it would be much harder to perform our attacks." This is exactly the method that überdiversity uses, meaning that its high entropy provides a native level of protection against these types of attacks.

Return-Oriented Programming. Return-oriented programming (ROP) [49] is a method that uses small bits of code known as "gadgets", each of which is a short sequence of instructions followed by a return. A carefully crafted stack can direct control flow through these gadgets in a particular order to preform arbitrary computation.

Traditional ASLR disturbs ROP by making it more difficult to locate gadgets. However, an attacker needs to find only one address range in order to locate all of the possible gadgets. Our prototype makes it even more difficult by putting each function, and therefore all potential gadgets, into unique random address ranges.

... and beyond In the time since the publication of ASLRSLR, several attacks against ASLR have been published. Some, such as [44], rely on the relatively predictable layout of the address space and are therefore defeated by using fine-grained randomization such as that explored in überdiversity. However, some state of the art techniques such as BROP [5] manage to defeat even fine-grained randomization without access to the compiled and loaded code. This powerful technique threatens otherwise secure randomization schemes, including überdiversity, but requires some particular properties of a system. The authors state that "the most basic protection against the BROP attack is to rerandomize canaries and ASLR as often as possible," because they rely on a restart property to guarantee that their addresses of interest do not change upon a process crash. Although überdiversity currently randomizes only during a call to `exec`, as they point out is typical of randomization techniques, it is possible that it could randomize during `fork` as well. The challenges of implementing a system to re-randomize code on a fork is discussed in the context of per-process kernel layout randomization in [7].

6 Conclusion

Diversity is a well-established technique for decreasing the risk of vulnerability amplification and increasing attacker workload in networked systems. Although many different techniques for injecting randomness into computer memory layouts have been presented, none have yet approached the limits of 64-bit hardware. Furthermore, most do not discuss the method used to randomize the memory layout, and many confuse entropy with program variants while attempting to quantify their effort. This paper presented überdiversity, an ELF diversity loader that randomizes fine-grained memory regions while loading them into memory.

A study of related work reveals that an important detail in any randomization technique, the algorithm used to produce permuted address spaces, is rarely mentioned, much less thoroughly examined. We present the algorithm we use and prove that it produces address space layouts uniformly at random, a property that is necessary for any randomization technique to be deployed most effectively. Additionally, previous research efforts are inconsistent in the use of "entropy" to quantify the effectiveness of a diversification tool. To address this, we take care to separate the measure of entropy from that of program variants; both provide valuable *but different* ways to estimate the potency of a diversity implementation.

Überdiversity makes several improvements over the current state of the art. Firstly, it manages to make use of a large majority of the available virtual memory, delivering an unprecedented 43-47 bits of entropy. Including the diversity loader into the system's bootloader enables the random loading of the hypervisor and the operating system kernel in addition to the more common randomization of user processes. Furthermore, the kernel and the user process are randomized together, producing an address space where kernel- and user-sections are non-deterministically interleaved. The flexible x86 virtual memory abstraction maintains appropriate protections despite having discontiguous kernel- and user-space virtual memory regions.

Our prototype revealed high costs (7.2×) at load-time but, despite clear evidence of an extremely slow worst-case scenario, only moderate costs in the average case at run-time (less than 10%). This suggests that mainstream ASLR implementations could use more thorough and fine-grained randomization techniques on an average application, even approaching the hardware-imposed limits, without suffering unreasonable performance costs.

Acknowledgments

Many thanks to Peter Horak, Ivan Antoniv, and Jeff "PKT" Bass for their help in developing the mathematical rigor presented in Sections 3 and 5.1.2; Per Larsen for feedback on an early draft of this paper and valuable input about the distinction of entropy and program varaints; Steve Chapin for feedback on a late draft of the paper; and Morgon Kanter whose Ph.D. thesis work pro-

vided the starting point for the überdiversity implementation.

This material is based on research sponsored by the Defense Advanced Research Projects Agency (DARPA) under agreement number FA8750-11-2-0257. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

References

[1] AIM Independent Resource Benchmark, February 2013.

[2] AVIZIENIS, A., AND CHEN, L. On the Implementation of N-version Programming for Software Fault Tolerance during Execution. In *Proc. the First IEEE-CS International Computer Software and Applications Conference (COM PSAC 77), Chicago* (1977).

[3] BACKES, M., HOLZ, T., KOLLENDA, B., KOPPE, P., NÜRNBERGER, S., AND PEWNY, J. 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* (New York, NY, USA, 2014), CCS '14, ACM, pp. 1342–1353.

[4] BHATKAR, S., SEKAR, R., AND DUVARNEY, D. C. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14* (Berkeley, CA, USA, 2005), SSYM'05, USENIX Association, pp. 17–17.

[5] BITTAU, A., BELAY, A., MASHTIZADEH, A., MAZIÈRES, D., AND BONEH, D. Hacking blind. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2014), SP '14, IEEE Computer Society, pp. 227–242.

[6] BROOKES, S., DENZ, R., OSTERLOH, M., AND TAYLOR, S. ExOShim: Preventing Memory Disclosure using Execute-Only Kernel Code. In *Proceedings of the 11th International Conference on Cyber Warfare and Security* (April 2016), ICCWS'16, pp. 56–66.

[7] BROOKES, S., OSTERLOH, M., DENZ, R., AND TAYLOR, S. The KPLT: The Kernel as a shared object. In *Military Communications Conference, MILCOM 2015 - 2015 IEEE* (Oct 2015), pp. 954–959.

[8] C0NTEX. Bypassing non-executable-stack during exploitation using return-to-libc. `http://www.open-security.org/texts/4`.

[9] CHEN, X., SLOWINSKA, A., ANDRIESSE, D., BOS, H., AND GIUFFRIDA, C. Stackarmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries. In *NDSS Symposium 2015* (San Diego, California, 2015).

[10] CHEW, M., AND SONG, D. Mitigating buffer overflows by operating system randomization. Tech. rep., 2002.

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

[12] COMMITTEE, T. Tool Interface Stadard (TIS) Executable and Linking Format (ELF) Specification, May 1995.

[13] CRANE, S., LIEBCHEN, C., HOMESCU, A., DAVI, L., LARSEN, P., SADEGHI, A., BRUNTHALER, S., AND FRANZ, M. Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy* (2015).

[14] DAVI, L. V., DMITRIENKO, A., NÜRNBERGER, S., AND SADEGHI, A.-R. 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* (New York, NY, USA, 2013), ASIA CCS '13, ACM, pp. 299–310.

[15] DURSTENFELD, R. Algorithm 235: Random permutation. *Commun. ACM 7*, 7 (July 1964), 420–.

[16] EDGE, J. Kernel address space randomization, October 2013.

[17] FELLER, W. *An Introduction to Probability Theory and Its Applications*, vol. 1. Wiley, January 1968.

[18] FERRIE, P. Attacks on more virtual machine emulators. *Symantec Technology Exchange* (2007).

[19] FISHER, R. A., YATES, F., ET AL. Statistical tables for biological, agricultural and medical research. *Statistical tables for biological, agricultural and medical research.*, Ed. 3. (1949).

[20] FORREST, S., SOMAYAJI, A., AND ACKLEY, D. Building Diverse Computer Systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)* (Washington, DC, USA, 1997), HOTOS '97, IEEE Computer Society, pp. 67–.

[21] GIONTA, J., ENCK, W., AND NING, P. HideM: Protecting the Contents of Userspace Memory in the Face of Disclosure Vulnerabilities. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy* (New York, NY, USA, 2015), CODASPY '15, ACM, pp. 325–336.

[22] GIUFFRIDA, C., KUIJSTEN, A., AND TANENBAUM, A. S. Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)* (Bellevue, WA, 2012), USENIX, pp. 475–490.

[23] HISER, J., NGUYEN-TUONG, A., CO, M., HALL, M., AND DAVIDSON, J. W. Ilr: Where'd my gadgets go? In *2012 IEEE Symposium on Security and Privacy* (May 2012), pp. 571–585.

[24] HOMESCU, A., NEISIUS, S., LARSEN, P., BRUNTHALER, S., AND FRANZ, M. Profile-guided Automated Software Diversity. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* (Washington, DC, USA, 2013), CGO '13, IEEE Computer Society, pp. 1–11.

[25] HUND, R., WILLEMS, C., AND HOLZ, T. Practical timing side channel attacks against kernel space aslr. In *Security and Privacy (SP), 2013 IEEE Symposium on* (May 2013), pp. 191–205.

[26] INTEL. *Intel 64 and IA-32 Architectures Software Developer's Manual*, June 2014.

[27] JACKSON, T., HOMESCU, A., CRANE, S., LARSEN, P., BRUNTHALER, S., AND FRANZ, M. Diversifying the software stack using randomized nop insertion. In *Moving Target Defense II*, S. Jajodia, A. K. Ghosh, V. Subrahmanian, V. Swarup, C. Wang, and X. S. Wang, Eds., vol. 100 of *Advances in Information Security*. Springer New York, 2013, pp. 151–173.

[28] JACKSON, T., SALAMAT, B., HOMESCU, A., MANIVANNAN, K., WAGNER, G., GAL, A., BRUNTHALER, S., WIMMER, C., AND FRANZ, M. Compiler-Generated Software Diversity. In

Moving Target Defense, S. Jajodia, A. K. Ghosh, V. Swarup, C. Wang, and X. S. Wang, Eds., vol. 54 of *Advances in Information Security*. Springer New York, 2011, pp. 77–98.

[29] KANTER, M. *Enhancing Non-determinism in Operating Systems*. PhD thesis, Thayer School of Engineering at Dartmouth College, October 2013.

[30] KANTER, M., AND TAYLOR, S. Attack Mitigation through Diversity. In *Military Communications Conference, MILCOM 2013 - 2013 IEEE* (Nov 2013), pp. 1410–1415.

[31] KEMERLIS, V. P., PORTOKALIDIS, G., AND KEROMYTIS, A. D. kGuard: Lightweight Kernel Protection Against Return-to-user Attacks. In *Proceedings of the 21st USENIX Conference on Security Symposium* (Berkeley, CA, USA, 2012), Security'12, USENIX Association, pp. 39–39.

[32] KIL, C., JUN, J., BOOKHOLT, C., XU, J., AND NING, P. Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software. In *Proceedings of the 22Nd Annual Computer Security Applications Conference* (Washington, DC, USA, 2006), ACSAC '06, IEEE Computer Society, pp. 339–348.

[33] LARSEN, P., BRUNTHALER, S., DAVI, L., AND SADEGHI, A.-R. *Automated Software Diversity*. Morgan & Claypool, 2015.

[34] LARSEN, P., HOMESCU, A., BRUNTHALER, S., AND FRANZ, M. Sok: Automated software diversity. In *Security and Privacy (SP), 2014 IEEE Symposium on* (2014), IEEE, pp. 276–291.

[35] LEVER, C., AND BOREHAM, C. malloc() Performance in a Multithreaded Linux Environment. In *Proceedings of USENIX Annual Technical Conference* (2000), pp. 55–56.

[36] LEVINE, J. R. *Linkers and Loaders*, 1st ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[37] LHEE, K.-S., AND CHAPIN, S. J. Buffer overflow and format string overflow vulnerabilities. *Software: Practice and Experience 33*, 5 (2003), 423–460.

[38] LOVÉN, T. Recursive page directory, June 2012.

[39] MARCO-GISBERT, H., AND RIPOLL-RIPOLL, I. Exploiting Linux and PaX ASLR's weaknesses on 32- and 64-bit systems. In *Blackhat Asia 16* (April 2016).

[40] MÜLLER, T. Aslr smack & laugh reference. In *Seminar on Advanced Exploitation Techniques* (2008).

[41] NICHOLS, C. Bear - a Resilient Core for Distributed Systems. Master's thesis, Thayer School of Engineering at Dartmouth College, 2013.

[42] NICOLE PERLROTH, J. L., AND SHANE, S. N.s.a. able to foil basic safeguards of privacy on web, September 2013.

[43] NOVARK, G., AND BERGER, E. D. Dieharder: securing the heap. In *Proceedings of the 17th ACM conference on Computer and communications security* (2010), ACM, pp. 573–584.

[44] OIKONOMOPOULOS, A., ATHANASOPOULOS, E., BOS, H., AND GIUFFRIDA, C. Poking holes in information hiding. In *25th USENIX Security Symposium (USENIX Security 16)* (Austin, TX, 2016), USENIX Association, pp. 121–138.

[45] OTTERSTAD, C. Brute force bypassing of aslr on linux. *Norsk informasjonssikkerhetskonferanse (NISK) 2012* (2012).

[46] PAPPAS, V., POLYCHRONAKIS, M., AND KEROMYTIS, A. D. Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2012), SP '12, IEEE Computer Society, pp. 601–615.

[47] PRESS, W. H. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, 2007.

[48] RANDELL, B. System structure for software fault tolerance. In *ACM SIGPLAN Notices* (1975), vol. 10, ACM, pp. 437–449.

[49] ROEMER, R., BUCHANAN, E., SHACHAM, H., AND SAVAGE, S. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Trans. Inf. Syst. Secur. 15*, 1 (Mar. 2012), 2:1–2:34.

[50] ROSCOE, T., ELPHINSTONE, K., AND HEISER, G. Hype and Virtue. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems* (Berkeley, CA, USA, 2007), HOTOS'07, USENIX Association, pp. 4:1–4:6.

[51] SHACHAM, H., JIN GOH, E., MODADUGU, N., PFAFF, B., AND BONEH, D. On the Effectiveness of Address-Space Randomization. In *In CCS 04: Proceedings of the 11th ACM Conference on Computer and Communications Security* (2004), ACM Press, pp. 298–307.

[52] SHIOJI, E., KAWAKOYA, Y., IWAMURA, M., AND HARIU, T. Code shredding: Byte-granular randomization of program layout for detecting code-reuse attacks. In *Proceedings of the 28th Annual Computer Security Applications Conference* (New York, NY, USA, 2012), ACSAC '12, ACM, pp. 309–318.

[53] SNOW, KEVIN Z. AND MONROSE, FABIAN AND DAVI, LUCAS AND DMITRIENKO, ALEXANDRA AND LIEBCHEN, CHRISTOPHER AND SADEGHI, AHMAD-REZA. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2013), SP '13, IEEE Computer Society, pp. 574–588.

[54] TEAM, P. Address Space Layout Randomization. `https://pax.grsecurity.net/docs/aslr.txt`, 2001.

[55] WARTELL, R., MOHAN, V., HAMLEN, K. W., AND LIN, Z. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 157–168.

[56] XU, H., AND CHAPIN, S. J. Address-space Layout Randomization Using Code Islands. *J. Comput. Secur. 17*, 3 (Aug. 2009), 331–362.

[Proofs Omitted from Section 3]

Proof of Non-Uniform Random Outputs of Algorithm 2

Proof. Line 2 of Algorithm 2 generates R, a set of $|S|+1$ random integers. Line 3 transforms R into T using a scaling routine, $f()$, so that the sum of the size of the sections plus the random numbers in T will fill the address space entirely. Then, the $|S|+1$ scaled random numbers in T become the gaps between the sections when they are loaded into memory.

The vital observation is that T to R is not a 1:1 mapping. If $f(R) = T$, then $f(2R) = T$, $f(3R) = T$, etc. However, each T results in exactly 1 possible address space. We will call $f^{-1}(T)$ the set of all sets of R such that $f(R) = T$. So, $f^{-1}(T) = \{R : f(R) = T\}$. Since all

Algorithm 2: Scaling Diversity Loader Algorithm. This algorithm chooses enough random numbers for each to be a gap between two sections, then scales the random numbers to fit the available address space.

Input: The Address Space A & Sections to be Loaded S
Output: Random location s_i.location $\forall s \in S$

Randomly shuffle S;
Choose $|S|+1$ random integers from 0 to RAND_MAX; $\{r_1, r_2, ..., r_{|S|+1}\} = R$;
$T \leftarrow f(R)$ where $f(R) = \left\lfloor R \left(\frac{|A| - \sum_{s \in S} s.\text{size}}{\sum_{r \in R} r} \right) \right\rfloor$;
s_1.location $\leftarrow t_1$;
for $i = 2$ **to** $|S|$ **do**
    s_i.location $\leftarrow s_{i-1}$.location $+ t_i$;

possible sets R are equally likely and each T generates a unique address space, uniform randomness implies that, given T, $\nexists T' : |f^{-1}(T)| \neq |f^{-1}(T')|$.

Unfortunately, this is not true. Intuitively, if a set of numbers in R is not big enough to fill the address space and needs to be scaled *up*, then the resulting numbers in T will be larger. In other words, the process of scaling means that address space layouts with small gaps between sections are less likely than those with large gaps between sections. More formally, consider a set T in which all members are even. In this set, $f^{-1}(T) \supseteq \{0.5T, T, 2T, 3T, 4T, ...\}$. However, if T' contains one or more elements of value 1, $0.5T' \notin f^{-1}(T')$ since we are dealing only with integers, and no element of an R that had to be scaled up could result in a value of 1. Therefore, $\exists T, T' : |f^{-1}(T)| \neq |f^{-1}(T')|$. By proof by contradiction, this algorithm does *not* produce address space layouts with uniform probability.

■

Proof of Lemma 1

Lemma 1. *Algorithm 3 will always succeed to load a set of sections S with size $S.size = \sum_{s \in S} s.size$ if, $|A| \geq 2|S| \times s_l.size$ where $\nexists s \in S : s.size > s_l.size$.*

Proof. The largest section in S, s_l (given by definition: $\nexists s \in S : s.\text{size} > s_l.\text{size}$), will consume at least s_l.size eligible addresses when it is placed. The concern is that the space between some hard boundary, like address `0x0`, and the location where s_l is loaded ($a_{l,k}$) is too small for any address within that range to be eligible to load a subsequent section. In other words, the worst case is that loading each section removes $2s_l.\text{size} - 1$ from the available address space. Therefore, for all $|S|$ sections, the maximum address space that could be consumed is $|S|(2s_l.\text{size} - 1) = 2|S| \times s_l.\text{size} - |S| < 2|S| \times s_l.\text{size}$. Therefore, a single contiguous address range of size $2|S| \times s_l.\text{size}$ will be sufficient to load any set of sections. ■

Proof of Lemma 2

Lemma 2. *The algorithm (FYS') that differs from FYS by iteratively choosing a "place in line" for each member in N instead of iteratively choosing a random member of N to place "next in line" produces variants uniformly at random.*

Proof. In each of FYS and FYS', a particular sequence of random numbers chosen during the algorithm corresponds to a single and unique permutation of the address space. Because the method of choosing random numbers is the same between algorithms, the probability of a given sequence of random numbers is the same in each. If the probability of any two variants is equal in FYS, then the probability of any two sequences of random numbers is equal. If the probability of any two sequences of random numbers is equal in FYS, then the probability of any two sequences of random numbers is the same in FYS'. Finally, if the probability of any two sequences of random numbers is equal, then the probability of any two variants resulting from FYS' is equal. Therefore, FYS' produces program variants with uniform and evenly distributed probability. ■