# Effectiveness of Moving Target Defenses

David Evans, Anh Nguyen-Tuong, John Knight

**Abstract** Moving target defenses have been proposed as a way to make it much more difficult for an attacker to exploit a vulnerable system by changing aspects of that system to present attackers with a varying attack surface. The hope is that constructing a successful exploit requires analyzing properties of the system, and that in the time it takes an attacker to learn those properties and construct the exploit, the system will have changed enough by the time the attacker can launch the exploit to disrupt the exploit's functionality. This is a promising and appealing idea, but its security impact is not yet clearly understood. In this chapter, we argue that the actual benefits of the moving target approach are in fact often much less significant than one would expect. We present a model for thinking about dynamic diversity defenses, analyze the security properties of a few example defenses and attacks, and identify scenarios where moving target defenses are and are not effective.

## 1 Introduction

The idea of security through diversity is to automatically generate variants of a target program or system that alter certain properties of the system. These alterations are designed to preserve the essential semantics of the original program on normal inputs, but to alter its behavior on malicious inputs. A widely deployed example is address space randomization, forms of which are included in most modern operating systems including Mac OS X, Ubuntu, Windows Vista, and Windows 7. Address space randomization thwarts exploits that depend on known absolute addresses for objects in memory by randomizing the locations of those objects. As we discuss in Section 4, although address space randomization does disrupt many attacks, it is vulnerable to brute force attacks because of the limited entropy used in many address space randomization implementations, and vulnerable to probing attacks.

University of Virginia
e-mail: [evans, nguyen, knight]@virginia.edu

Moving target defenses seek to overcome the limitations of static diversity defenses by dynamically altering properties of programs. For long-running server processes, this requires dynamically altering the running execution (or, more disruptively, periodically restarting the process with a new randomization). If the attack surface changes rapidly enough, the hope is that dynamic diversity defenses can protect systems even in situations where static diversity would be vulnerability to low entropy or probing attacks.

In this chapter we develop a model for moving target defenses, and analyze their effectiveness against sophisticated attackers. We argue that in many cases the added security a dynamic diversity defense provides against such attackers is quite limited and can be quantified. In other scenarios, where there are good reasons to believe the time required to develop an effective exploit is high, dynamic diversity defenses can provide significant benefits over static diversity.

## 2 Diversity Defenses

The goal of a diversity defense is to present attackers with an unpredictable target, thereby making it difficult for an exploit to have the desired malicious behavior. Diversity techniques may be applied at a low-level, where the standard semantics of the programming language are preserved but its undefined semantics altered. This has the advantage that it can be done automatically, without needing any behavioral specification of the target program other than belief that its behavior does not depend on undefined language semantics. The limitation of such low-level diversity techniques is that they can only change behavior for exploits that exploit the altered undefined semantics. This covers many important classes of attacks including most code injection and memory corruption attacks, but does not include any attacks that exploit the application's higher-level semantics.

The other type of diversity defense attempts to alter that applications' higher-level behavior. This depends on a sufficiently clear understanding of the application's required behavior to be able to alter the application's semantics in ways that may disrupt attacks but do not impact its essential functionality. The drawback of higher-level diversity defenses is that they typically require manual effort to produce the variants, and because they are constructed in ad hoc ways it is much more difficult to reason about the security they provide. It is also difficult to use such an approach in a dynamic diversity scenario since it requires a large number of variants to provide a moving target.

In this chapter, we focus on low-level, automatic, diversity defenses. The idea of automatically generating diverse variants of a program to disrupt exploits was introduced by Forrest et al. [32], and many subsequent works considered various ways for automatically generating useful diversity in program executions. Here we describe three common types of automatic diversity techniques. Although the model and analysis we present applies to a wide range of diversity techniques, our examples focus on the most commonly used techniques described here.

## 2.1 Address Space Randomization

Address space randomization or *address space layout randomization* (ASLR) is the most successful and widely deployed diversity technique. The basic idea is simple: randomize the locations of objects in memory so an attack that depends on knowing the address of these objects will fail. Address space randomization was first implemented by PaX for Linux [35] in 2000, and has since been implemented in most major operating systems including Windows (first in Windows Vista in 2007, and later in Windows Server 2008 and Windows 7), Linux (partially included in the Linux kernel since 2005, and more complete implementations in most hardened Linux distributions), and Mac OS (in a limited form since OS X 10.5).

The simplest ASLR implementations just randomize the base address for large memory areas. For example, PaX randomizes the base addresses for the executable area containing the program's code and static data structures, the stack area containing the execution stack, and the mapped memory area containing the heap as well as shared memory and dynamically-loaded libraries. The address of each of these areas is randomized by adding a randomly generated offset to the address. Within each area, though, the layout is unchanged. The advantage of such an approach is it can be implemented by the loader without any changes needed to the executable.

Other implementations of ASLR more comprehensively randomize the address layout. For example, address obfuscation randomizes the both the absolute locations of data and code as well as their relative locations [3]. This can be done by randomly permuting the order of variables on the stack or in a structure, as well as by adding random padding between objects. Unlike randomizing segment base addresses, however, making such changes requires deeper analysis of the target program.

## 2.2 Instruction Set Randomization

Instruction set randomization is a general technique for thwarting code injection attacks by obscuring the instruction set of the target [14, 12, 13]. An attacker who knows an exploit that allows code constructed by the attacker to be injected into the target application will not be able to create code that has the desired behavior without knowing the target instruction set.

An example implementation of instruction set randomization is Barrantes et al.'s RISE [12]. The instruction set is randomized by generating a sequence of random bytes and XORing each instruction in the program with a corresponding random byte when the code text is loaded. Then, the program is executed in an emulator that XORs the instruction with the random byte to obtain the original instruction. A code injection attack that does not know the randomization key will not be able to generate the desired behavior, since the injected instructions will be XORed with random bytes before they are executed.

Other implementations of instruction set randomization us block encryption instead of bytewise XORs. For example, Hu et al. implemented a form of instruction set randomization by encrypting program code with AES at the granularity of 128-bit blocks [36]. Higher-level instruction sets can also be randomized. For example, SQL injection attacks can be thwarted by adding random nonces to SQL commands [33] and Perl injection attacks can be thwarted by randomizing parts of the Perl language [4].

### *2.3 Data Randomization*

Another type of low-level diversification is altering how data is stored in memory. An early instantiation of this idea was PointGuard [8], which attempts to thwart pointer corruption attacks by storing pointers in memory XORed with a random key. When a pointer value is loaded into a register, it is XORed with the key to produce the actual pointer value. A more general technique we developed by Cadar et al. [10]. They XORed data in memory with random masks, selected based on the memory object's class. This requires a static analysis of the program to determine memory regions that are associated with particular objects, so that attempts to write outside objects will be disrupted since different random masks are applied.

## 3 Model

We consider a model involving two players: an attacker and a defender. The defender's goal is to provide a service, $S$, with a high reliability and performance. The attacker's goal is to successfully exploit the server. We assume the service has at least one vulnerability which is known to the attacker but not to the defender. An attacker with knowledge of the full state of the system can launch an exploit that compromises the server. We define $t_e$ as the time between starting to launch the exploit and the system compromise. For our purposes, it is not necessary to specify the actual harm the compromise causes, but we can think of this as obtaining confidential account information from the server.

In a static diversity defense, instead of running $S$, the defender generates a random secret key, $k \in K$, and executes $S_{\mathscr{T}_k}$ where $\mathscr{T}$ is a key-dependent transformation. The transform preserves the essential semantic of $S$; that is, for all legitimate inputs $x \in \mathscr{N}$, $S(x) \approx S_{\mathscr{T}_k}(x)$, where $\approx$ indicates a loose semantic equivalence test that may be service-specific. The intent of the transformation is to alter the service's response to attack inputs. For a particular targeted class of attack inputs, $a \in \mathscr{A}$, $S(a) \neq S_{\mathscr{T}_k}(a)$. In particular, while $S(a)$ constitutes compromise behavior, $S_{\mathscr{T}_k}(a)$ is harmless behavior.

An attacker who can determine $k$, or possibly only determine some information about $k$, can construct an exploit $a_k$ that achieves the desired compromise:

$S(a) \approx S_{\mathscr{T}_k}(a_k)$. Thus, the diversity defense succeeds when all the attacker's exploits are in the target class of attack input $\mathscr{A}$ and the attacker has not learned enough information about $k$ to construct the exploit $a_k$.

Consider the lifetime of a particular service, shown in Figure 1. At time $t_0$, the defender has generated a random key $k_0$ and launches the diversified service, $S_{\mathscr{T}_{k_0}}$. The attacker knows a vulnerability in $S$ and an attack $a \in \mathscr{A}$ that exploits that vulnerability but is thwarted by the diversification. Starting at time $t_0$, the attacker attempts to exploit the running service. This may be done by generating variants of $a$ transformed around guessed randomization keys. It may also involve sending probe packets designed to leak information about $k$.

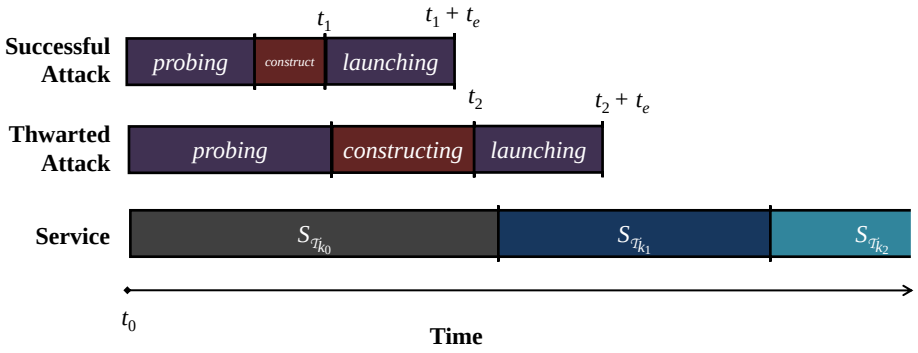


**Fig. 1** Attack Lifetime

The probability of the attack succeeding is a function of the amount of information the attacker has obtained about $k$. Assuming a simple defense where the attacker needs to guess all bits of $k$ completely for the attack to succeed, but has zero probability of success otherwise:

$$Pr[S_{\mathscr{T}_k}(a_{k_g}) \approx S(a)] = Pr[k_g = k] \tag{1}$$

If the attacker has no information about $k$ but must guess $k$ exactly to construct a successful attacks, when $|K| = 2^N$ (that is, $k$ has $N$ bits of entropy) $Pr[k_g = k] = \frac{1}{2^N}$.

The attacker may be able to obtain information about $k$ by sending probes. This increases $Pr[k_g = k]$ over time as the attacker learns more about the target service. At some later time, $t_1$, the attacker finds a successful exploit against $S_{\mathscr{T}_{k_0}}$ and launches that exploit against the service. This exploit compromises the service at time $t_1 + t_e$.

If dynamic diversity is employed, the service is periodically rediversified with a new key. If that transformation happens during the probing phase or the exploit execution phase, it changes the target system to a new target $S_{\mathscr{T}_{k'}}$. This disrupts the attack $a_{k_g}$ since although $k_g = k$ the system is now diversified with $k' \neq k_g$. In the case of the second attack, $t_2 + t_e$ is past when the service has been rediversified, so

although the constructed attack would succeeded if it had been launched at time $t_1$, it fails when it is launched at time $t_2$.

Our goal is to understand what types of diversity and attacks can be disrupted by such a strategy, and how the rate if re-diversification impacts the attacker's success probability.

## 4 Attack Strategies

The effectiveness of a moving target defense depends on the attacker's capabilities, resources, and strategy. Here, we consider several different broad strategies an attacker may employ against diversity defenses. In Section 5, we consider how much additional advantage dynamic diversity provides against each attack strategy. Note that we do not consider denial-of-service attacks here. Low-level diversity defenses often turn code injection or memory corruption attacks into denial-of-service attacks, which are generally less harmful than injection and corruption attacks since they do not expose or compromise any confidential data. Hence, although denial-of-service is undesirable, we consider it a successful attack disruption if an attack that would normal corrupt or compromise data is transformed into a denial-of-service by the diversity defense.

### *4.1 Circumvention Attacks*

The first attacker strategy is to circumvent the diversification entirely. This can be done if the attacker finds any exploit that does not depend on the properties of the server that are altered by the diversification. For example, an attacker may be able to circumvent a instruction set randomization defense by avoiding the need to inject code. Instead, the attacker repurposes code already provided by the executing binary. An early form of this strategy is the *return-to-libc* attack [1], in which the attacker replaces the return address on the stack with an address to an exploitable function in libc and loads the appropriate arguments on the stack. Shacham et al. introduced a more general form of this attack strategy known as *return-oriented programming*. Instead of relying on the functions provided intentionally by libc, return-oriented programming exploits fragments of code found in the binary (including fragments that start in the middle of intended instructions) to provide a Turing-complete programming system without needing to inject any code. A recent exploit against Adobe Reader/Acrobat used return-oriented programming to circumvent ASLR in Windows 7 and Windows Vista [21].

Another type of circumvention attack exploits incomplete randomization. For example, the Mac OS X Snow Leopard implementation of ASLR randomizes libraries but does not apply any randomization to the stack, heap, or program code [6]. An attacker can exploit a vulnerability in a program by taking advantage of non-

randomized portions of memory. The Windows 7 and Ubuntu implementations of ASLR randomize the operating system components completely, but only randomize the program image when developers set the appropriate (non-default) compiler flag. For Ubuntu, this is due to the relatively high performance overhead of position independent executables on 32-bit architectures, as well as uncertainty about compatibility with all programs. Hence, only certain programs included in Ubuntu that are deemed to be security critical are compiled as position independent executables, and other programs are executed without randomizing the program image [18]. Müller provides examples of many forms of circumvention attacks against PaX ALSR that find ways to return into non-randomized portions of memory including the program text, static variable storage (BSS), and heap [20].

Another example of a circumvention attack that exploits incomplete randomization is to alter an exploit to depend only on the local relative addresses instead of global addresses. Standard ASLR implementations may change the absolute address of a target memory location, but not its relative position to some other objects. For example, if the value an attacker wants to corrupt is a field in a structure, it may be possible to overwrite this value by exploiting a buffer overflow vulnerability on a buffer that is stored as a different field in the same structure. It is not necessary for the attacker to know the absolute address of either object, only to know their relative locations. Some proposed implementations of ASLR do provide randomization at this level such as Bhatkar et al.'s [3], but it is not done by standard implementations and cannot be done safely in general without a deeper analysis of the program.

Finally, an attacker may circumvent randomization defenses by exploiting the program at a higher semantic level that is not effected by the randomization. For example, randomizing the instruction set and address space layout of a web server provides no mitigation against a SQL injection attack that is exploiting vulnerabilities in the high-level application logic. Randomizing the instruction set to prevent code injection provides no defense against memory corruption attacks that do not need to inject any code such as the attacks describe by Chen et al. [7].

## *4.2 Deputy Attacks*

In a *confused deputy attack* [16], an attacker finds a way to use a benign program in a malicious way. For randomization defenses, the main fear is that an attacker will be able to find a way to use the program to apply the randomizing transformation to the attacker's data.

For many diversity defenses, the randomization transformation is done at runtime by the program itself. Hence, the code to perform the transformation (and the randomization key) is present somewhere in the running program.

One attacker strategy avoids the need to break the diversification entirely. Instead, the attacker either finds a way to exploit the target system that does not depend on any properties altered by the diversification, or finds a way to deputize code included

in the executing program that performs the transformation to transform the injected attack.

An attack that is somewhat like a deputy attack is a partial overwrite attack. Unlike a deputy attack which repurposes existing code to launch an attack, the partial overwrite attack coopts existing data. Consider a program that is protected by a coarse-grained variant of address-space randomization. A partial overwrite attack that modifies the least-significant byte of an address $A$ so that the program transfers control flow to a targeted function $F$ would bypass any protection afforded by address-space randomization. The address of the targeted function, while randomized, would still be at a known offset from $A$. Durden descries a partial overwriting attacks against PaX ALSR [11].

### *4.3 Brute Force and Entropy Reduction Attacks*

A *brute force attack* simply attempts all possible randomization keys until an exploit is found that succeeds. If the key space is small enough, such an attack may be practical. For example, Shacham et al. demonstrated an effective brute force attack against an Apache server protected using PaX ASLR [29]. A 32-bit architecture provides at most 32 bits of entropy for address randomization, but because of limitations on address mapping that actual entropy provided by PaX is only 16 bits for the executable and memory mapped areas, and 24 bits for the stack. Since the shared libc library is stored in a memory mapped area, is it only necessary to search 16 bits to locate the library and launch a return-to-libc attack. On average, their attack succeeds against a vulnerable Apache server in approximately 216 seconds on average.

For larger key spaces, attackers may find ways to reduce the effective key space by designing attacks that work for a set of possible keys. This changes the success probability in the original model from Equation 1 to:

$$Pr[S_{\mathscr{T}_k}(a_{k_g}) \approx S(a)] = Pr[k \in \mathscr{W}_a(k_g)]$$

where $\mathscr{W}_a(k)$ is the set of keys that are equivalent to $k$ with respect to attack $a$. The attacker's goal is to construct an attack $a$ for which

$$\bigcup_{k_g \in \mathscr{G}} \mathscr{W}_a(k_g) = K$$

for the smallest possible set $\mathscr{G}$.

A longstanding example of an entropy reduction attack is a *NOP sled*, widely used in standard stack smashing buffer overflow attacks to overcome uncertainty about memory layout even without the use of ASLR. With a NOP sled, the attacker inserts a series of one-byte NOP (No Operation Performed) instructions before the attack code. To avoid intrusion detection systems that alert on suspected NOP sleds, attackers can use other instructions that have no or limited sematic impact, or se-

quences of multi-byte instructions that can be interpreted as NOPs starting at any of their bytes [25, 24].

Since each instruction in the NOP sled is choosen to have no semantic effect, if the attacker can redirect execution to jump to any location in the NOP sled it will have the same behavior as jumping to the specific location where the attack code begins. The longer the NOP sled, the higher the probability a jump to a randomized location will land within the NOP sled and reach the attack code. For example, if a 127-byte NOP sled is used, $|\mathscr{W}_a(k_g)| \approx 128$, effectively reducing the randomization entropy by up to 7 bits (the actual reduction is probably less, for example, if the randomization offsets must be word-aligned).

A more extensive form of entropy reduction is *heap spraying* in which an attacker attempts to fill up a large fraction of memory in a way that increases the likelihood of reaching a target object. An early example of heap spraying was Govindavajhala and Appel's attack to circumvent type safety mechanisms on Java virtual machines [15]. The attack was not designed to overcome intentional address space randomization, but instead to take advantage of random bit errors (caused, by example, by heating up memory until there is a high liklihood of single bit errors).

Several recent attacks have used heap spraying from JavaScript to launch attacks on ASLR-protected web browsers [38, 19]. In a JavaScript heap spraying attack, the attacker uses JavaScript code executed by the browser to allocate a large number of objects in the heap [26]. Each object is constructed to include a NOP sled, followed by the attack code. This increases the likelihood that a jump to a randomized address will reach one of the copies in memory of the exploit code. A sophisticated version of the attack known as *heap feng shui* takes advantage of the way the heap allocator and garbage collector work to control more of memory and how the attack objects are arranged [30].

The effectiveness of randomization defenses is severely reduced by these types of entropy reduction attacks. In many cases, a well constructed heap sprying exploit succeeds on the first attempt with high probability.

## *4.4 Probing Attacks*

A probing attack attempts to overcome a diversity defense by using probe packets to learn properties of the randomized execution needed to construct an attack. A probe attack is distinguished from a standard entropy reduction attack in that the probe packets are designed only to obtain information about the target, rather than to produce the desired malicious behavior.

Shacham et al.'s attack on ASLR used probes to find the randomization offset for the memory map region, which could then be used to learn the locations of all libc functions and construct the attack [29]. The probe packets attempted to find the usleep function in libc by jumping to randomized addresses. The remoted attacker could observe when the usleep function was found since the call to usleep causes the connection to hang; if the guessed address is incorrect the server child process will

(most likely) crash. Once the usleep address is obtained, the attacker has enough information to compute the address of all the other libc functions, including the system function used to obtain a shell. In this case, probing does not have much advantage over just sending the guessed attack directly (that is, it is not any easier to guess the location of usleep than it is to guess the location of system), but does enable an attacker to use smaller, possibly harder to detect, packets to probe the system to learn the randomization key rather than needing to send the full attack payload with each guess attempt.

For the previous example, the amount of information the attacker receives for each probe attempt is very limited — if the guess is incorrect the server crashes and the attacker learns nothing more other than that this guess was not the correct offset. In some cases, though, probe attacks may be possible where each probe obtains a great deal of information. It may be possible to use information contained in server error messages returned to the attacker to learn addresses, or to exploit a format string vulnerability to obtain the address of a targeted object. Müller provides two examples based on pointer redirecting to obtain addresses of randomized functions [20].

Strackx et al. developed *buffer overread attacks* to expose randomized addresses in memory [34]. The attack takes advantage of a property of the strncpy library function, as well as other similar functions in the standard C library. These functions take a size parameter indicating the size of the output buffer to protect against buffer overflows, but do not automatically add a null terminator at the end of the result if the string being copied exceeds the size of the output buffer. Thus, when the string is printed, it will contain subsequent data in memory until the next NULL byte. This data may contain addresses, revealing the actual locations of randomized addresses. Similar attacks are also possible against instruction set randomization [37].

## *4.5 Incremental Attacks*

An increment attack is a form of probing attack where more than one successful probe is needed to obtain sufficient information to construct the exploit. For example, this occurs when the randomization key is many bytes long, but each successful probe packet obtains only a single key byte. This is the case for implementations of instruction set randomization that use an XOR mask to randomize the instructions. In Kc et al.'s proposed hardware design, the XOR mask is a four-byte value that is stored in a dedicated register [14]; in Barrantes et al.'s software implementation, RISE, the XOR mask can be as long as the program text [12].

Sovarel et al. developed an incremental attack against instruction set randomization [31]. The attack uses probes to determine key bytes until a large enough region of key bytes is known to inject the attack code. In one version of the attack, a single byte instruction (the 0xc3 return instruction) is guessed, and for some vulnerabilities it may be possible to incrementally break the randomization key one byte at a time. For most vulnerabilities, though, the difference in behavior from a correct and

incorrect guess of the return instruction is indistinguishable (both are likely to cause the server to crash, since the return instruction leaves the stack in a corrupted state). An alternate attack uses the two-byte short jump instruction with offset -2 (0xebfe) which jumps back to itself, causing the server to enter an infinite loop which can be distinguished by the attacker from the crash that usually results from an incorrect. By this method, a many byte key can be broken incrementally in two-byte chunks.

## 5 Analysis

The value of a moving target defense depends on the class of attack. For each attack strategy described in Section 4, we consider the impact of dynamic diversity over static diversity. In the first three cases, dynamic diversity appears to have little benefit; for incremental probing attacks, however, the situation is more interesting and dynamic diversity appears to have substantial value.

### *5.1 Circumvention Attacks*

In a circumvention attack, the transformation $\mathcal{T}_k$ does not diversify any aspects of $S$ that is required for a successful attack. Hence, there is no benefit to changing the diversification. Since the diversity transformation does not cover the attack class, reapplying the transformations with varying keys yields no benefits.

### *5.2 Deputy Attacks*

In a deputy attack, an attacker is able to induce the target program to apply the diversity transformation to the attack input. If the diversification key changes, this has no impact on the attack since the attacker is exploiting the actual transformation code in the program. Similarly to circumvention attacks, dynamic diversity provides no advantage for deputy attacks.

### *5.3 Brute Force and Entropy Reduction Attacks*

As noted in Section 4.3, Shacham et al. showed that PaX ASLR provides only enough entropy to slow the spread of a worm. Dynamic diversity provides modest benefit against a brute force or entropy reduction attack. Even if the target is rerandomized after every attack attempt, the maximum impact on the attacker is changing the random search from random sampling without replacement to random

sampling with replacement. This adds at most a single bit of entropy to the search space.

Hence, dynamic diversity provides little benefit as a defense mechanism against these attacks above and beyond the baseline static version. If the effective entropy of a diversity transform is low, the expected time to mount a successful attack would be relatively short, and therefore a factor of two would be of little value. When entropy is high, the expected time to mount a successful attack would be long, and again a factor of two provides limited additional benefit.

As we discuss in Section 6.1, if multiple diversity techniques are composed in a way that requires an attacker to incrementally break each of them, there may be more substantial gains possible by dynamically re-diversifying each technique in an interleaved way.

## *5.4 Probing Attacks*

Dynamic diversity could be useful against a probing attack if the time between a successful probe and completing an exploit is long. In practice, however, exploits can be constructed automatically based on the probe information, so the time between the probe and attack launch is effectively just the network latency for two round trips between the server and attacker's client. If the re-randomization can be done frequently enough, it may be possible to ensure that the system has always been re-randomized between the probe and exploit. Such frequent re-randomization is too expensive for most services, but perhaps could be done in some scenarios. This would be most effective against probing attacks such as the buffer overread attack that depend on using a first request to leak information about the randomization, and use that information to construct a targeted attack.

## *5.5 Incremental Attacks*

Dynamic diversity seems most promising against incremental attacks since these attacks require a lot of preparation by the attacker before enough information about the randomization is obtained to construct the attack. Here, we develop a model to analyze the effectiveness of dynamic diversity against incremental attacks. Our model applies to the Sovarel et al. attack described in Section 4.5, but should also apply to any incremental attack that involves sending a large number of probe packets to gradually acquire information about the randomization.

We model an incremental attack as a series of $b$ state-space searches where the states are of the same size $s$. A state-space search is carried out as a series of probes, and each state-space search is designed to reveal a single key fragment. Hence the key length is $b$ fragments. We define a successful attack as a sequence of successful state-space searches of the $b$ spaces.

We assume that:

- the adversary proceeds sequentially from space to space determining one fragment for each space,
- the adversary knows when a fragment has been revealed, and,
- each probe of a space requires the same time.

These assumptions simplify the analysis, but do not meaningfully limit the class of incremental attacks.

In this case, the quantity of interest is the probability of a successful attack occurring in some specific number of probes, say $L$, or less. With that probability known, re-randomization could be triggered after the adversary had an opportunity to perform $L$ probes and thereby limit the probability of a successful attack. Thus, our first goal in the analysis is to determine this probability. Clearly, we cannot know how many probes have occurred, but we can estimate the number of opportunities that the adversary had.

Searching each space will terminate with a successful probe, and each successful probe will be preceded by from zero to $s-1$ probes that fail. The initial step in the model is to determine the probability of a successful attack in exactly $L$ probes. Such an attack will experience a total of $k-b$ probe failures across all $b$ spaces together with $b$ successful probes. Thus, the total number of different sequences of probes that can lead to a successful attack in $L$ probes is the number of ways that $L-b$ failing probes can be distributed across $b$ spaces with no more than $s-1$ occurring in any single space. This number can be dervied using the *Balls In Bins* analysis [5]:

$$N(L) = \sum_{t=0}^{b} (-1)^t \binom{b}{t} \binom{L-ts-1}{b-1}$$

In this expression, binomial coefficients are defined to be zero if the upper operand is smaller than the lower operand.

The probability of a successful attack occurring in exactly $L$ probes for $b \leq L \leq sb$ is:

$$p(L) = \frac{N(L)}{2^{sb}}$$

The probability of a successful attack occurring in $L$ probes or less for $b \leq L \leq sb$ is:

$$P(L) = \sum_{i=b}^{L} p(i)$$

This is the probability we sought, and with this probability we can determine the effect of periodic re-randomization as a defense against an incremental attack.

We model the effect of re-randomization by treating an attack as a series of independent trials by the adversary each of length $m$ probes where the key is changed after each trial, i.e., after $m$ probes. Thus, the effect of re-randomization is to force

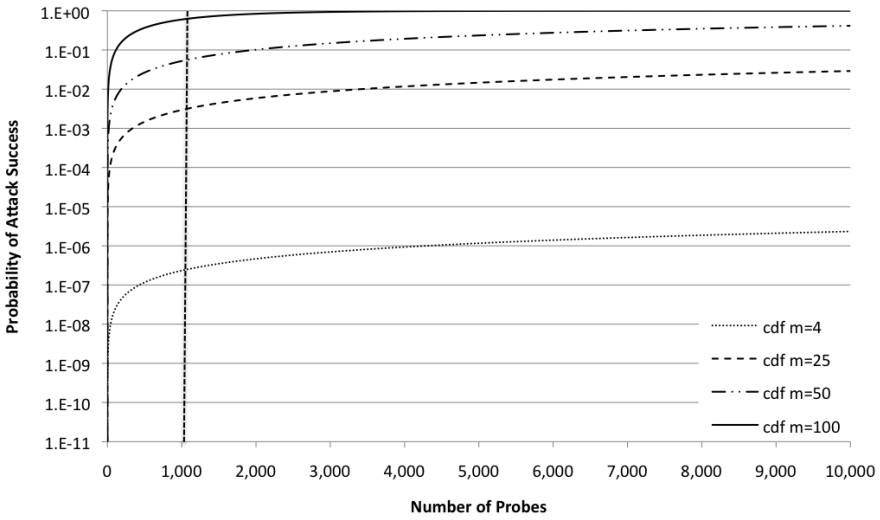


**Fig. 2** Effectiveness of dynamic diversity

the adversary to restart the attack after each series of $m$ probes if the attack was not successful at that point.

The probability of a successful attack in $m$ or fewer probes is $P(m)$. With re-randomization after each trial (each $m$ probes), the probability of an attack succeeding in $jm$ probes or fewer is:

$$M(jm) = 1 - (1 - P(m))^j$$

Note that this probability is defined only for every $m$ probes. In order to derive the probability of a successful attack in $L$ or fewer probes, we need to add the probability of a successful trial (determining a single fragment of the key) in $L - jm$ probes where $L - jm$ lies between 0 and $m - 1$, i.e., between the points at which the key is changed. Adding this yields:

$$M(L) = 1 - (1 - P(m))^j (1 - P(L - jm))$$

$M(L)$ is the probability of a successful attack in $L$ or fewers probes with re-randomizing every $m$ probes and $P(L)$ is that probability without re-randomization. With these two probabilities, we can determine the effectiveness of dynamic diversity.

As an example, consider the case in which $b = 4$ and $s = 256$. This corresponds to a key that is four bytes long which would be expected to have a search space of size $2^{32}$. However, the incremental attack proceeds one byte at a time so that there are four searches each of spaces of size 256. Obviously, the probability of success in 1024 probes or less is one.

Figure 2 shows the probability of attack success, $M(L)$, for values of $m$ = 4, 25, 50, and 100. Note that the Y axis is a logarithmic scale. The dashed vertical line is 1024 on the X axis. This is the point at which an attack is expected to succeed without dynamic diversity, and the intersection of the dashed line with the four curves shows the relative advantage of re-randomization. The case in which $m$ is set to 4 is the limiting case in this example. Four is the least number of probes within which an attack might succeed since there are four bytes in the key and the adversary has to determine all four in sequence. Thus, the curve in Figure 2 for $m = 4$ is the best that dynamic diversity can do in this example.

The effectiveness of dynamic diversity against incremental attacks in this case depends critically on the rate of re-randomization. Varying this rate from every $100^{th}$ probe to every $4^{th}$ probe spans 6 orders of magnitude. Moreover, the probability of attack success when re-randomizing only every $100^{th}$ or $50^{th}$ probe quickly exceeds 90%, i.e., dynamic diversity in these cases is ineffective.

For a server responding to network inputs, a network message corresponds to a probe in the model. Re-randomizing on every $4^{th}$ or $25^{th}$ network messages would seem prohibitively costly. However, our results show that it is possible to re-randomize XOR keys used in instruction-set randomization at the rate of every 100 ms with an average cost of 14% overhead over native execution [23]. Re-randomization may be triggered based on events instead of the current time-based scheme. For example, many probe attacks result in process crashes when the guess is incorrect, so it makes sense to rerandomize after a crash. Rerandomization may also be triggered by particular system calls (e.g., opening a file) or when an anomaly detector flags a packet as suspicious. The risk in any event-based rerandomization scheme is that a sophisticated attacker may be able to develop a probe attack that does not produce the trigger event. Hence, some combination of time-based and event-based rerandomization seems most promising.

## *5.6 Summary*

Table 1 summarizes the effectiveness of dynamic diversity against the five attack classes. For circumvention and deputy attacks, dynamic diversity yields no benefit since the attack does not depend on guessing the randomization key. For brute force and entropy reduction attacks, the benefits of dynamic diversity are marginal and only increase the attacker's workload by at most a factor of two. Dynamic diversity holds the most promise for probing and incremental attacks. The rate of re-diversification required to obtain tangible benefits, especially against probing attacks, appears to be very high, but for some types of implementation it may be possible to achive such a high rate of re-randomization without excessive performance overhead [23].

| | |
|---|---|
| Circumvention attacks | No advantage |
| Deputy attacks | No advantage |
| Entropy reducing attacks | At most doubles expected attack time |
| Probing attacks | Very high rate of rerandomization may thwart attack |
| Incremental attacks | May provide significant advantage |

**Table 1** Impact of Dynamic Diversity

## 6 Discussion

The limited effectiveness of adding dynamic rerandomization to low-level diversity defenses suggests the need for alternate approaches to increase the effectiveness of diversity defenses. Some of these depend on designing implementations to maximize entropy and avoid vulnerabilities, but since the goal of these defenses is to harden systems that have unknown vulnerabilities it is unsatisfying to rely solely on this approach. In addition, for schemes like adress space randomization the maximum amount of entropy available is limited by properties of the hardware and underlying operating system. Next, we discuss two approaches that can improve the effectiveness of diversity defenses. The first, composition, amplifies the value of rerandomization; the second requires an attacker to simultaneously compromise multiple variants, avoid the need to keep any secret key.

### *6.1 Composition*

Our analysis so far assumed a single diversity defense was deployed, and an attacker who can overcome that defense will succeed. One way to sbstantially increase the attacker's difficulty is to compose multiple diversity defenses. If the defenses are orthogonal, they will compose multiplicatively not additively. That is, it will be necessary for the attacker to simultaneously break both defenses, so the effective search space is the product of each defense's search space individually. This assumes the attacker cannot probe each defense separately, making the attack difficulty the sum of the two defenses. Worse, if the composition is not done in a careful way, the composed defense may provide the attacker with new opportunities that would not be effective against either defense individually.

Address space randomization and instruction set randomization can be composed. This would thwart many attacks on instruction set randomization, since the attacker cannot probe for the ISR key without also knowing the target address. If either the key or target address is incorrect, the server is likely to crash and the attacker does not learn if either key guess by itself is correct. Simplistically, if the address space randomization has 24 bits of entropy and the attacker can use a one-byte incremental probing attack, the combined defense provides 32 bits of entropy. On the other hand, combining address space randomization with instruction set ran-

domization does not provide any benefit against attacks such as return-oriented programming that do not need to inject code. The problem for the attacker is the same as with address space randomization alone, since there is no need for the attacker to break both defenses.

If defenses can be composed multiplicatively, however, many low-entropy defense may be combined to provide a high-entropy defense. Holland et al. proposed schemes for randomizing many properties of an execution using a virtual machine [17]. Many of the individual diversification strategies provided little entropy (for example, altering the number of registers and machine word size), but they argued that by composing them they could provide a large machine space where attackers may need to guess all the diversification parameters.

It is difficult to reason precisely about the orthogonality and multiplicity of a composition of diversity defenses, but it is a promising way to increase the entropy facing an attacker. Adding dynamic rerandomization to composed defenses is promising, since if the composition does have the property that an attacker much simultaneously break all of the diversity techniques, it is only necessary to schedule the rerandomizations of each defense in a tiled way to limit the amount of time all of the diversity parameters are unchanged.

## *6.2 N-Variant Systems*

Whereas composition strategies attempt to require the attacker to simultaneously break multiple different diversity defenses, the *N-Variant Systems* approach is designed to require an attacker to simultaneously break multiple variants of the same diversity defense [9]. The idea is to run multiple instantiations of the server in synchrony, each of which is diversified using a different randomization key. The variants are run in a framework that sends the same inputs to each variant, and monitors that they behave similarly. Any divergence considered a signal of a possible attack, since the variants should behave identically on non-attack inputs. This requires that the variants are kept closely synchronized and that any other sources of nondeterminism are removed.

If the attack spaces for the variants are disjoint with respect to some attack class, then no single input can simultaneously compromise all the variants. A simple example is to use two variants with disjoint address spaces (for example, one variant has only addresses beginning with a `0`, and the other variant has only addresses starting with a `1`). Then, any attack that depends on injecting an absolute address must fail — there is no address that is simultaneously valid in both variants. Several opportunities for disjoint attack spaces are possible including address spaces [9], instruction sets [9], and how data is represented [22]. When fully disjoint attack spaces cannot be found, a similar approach can also be used probabilistically. Examples include changing the direction of the stack [27] or diversifying the relative positions in memory (as done by DieHard [2], which focused on software debugging rather than attack detection).

A key advantage of this approach (at least for disjoint attack spaces) is that it eliminates the need to keep any secrets at all. This means brute force, entropy reduction, probing, and incremental attacks all fail. The main remaining worries are circumvention attacks, which are possible against any diversity defense if the diversification does not impact properties needed by the exploit, and deputy attacks, especially since each variant includes its own derandomization code or differently randomized data so may be simlutaneously exploited across the variants. There are, however, a number of challenges to deploying N-Variant systems in practice. The first is the close monitoring requires eliminating all causes of nondeterminism. This is particularly difficult in multi-threaded applications where the interleaving of threads may lead to divergence. Performance overhead is also a concern, since the approach requires duplicating each request. This overhead can be fairly low for I/O bound servers [9], and it may be further reduced by using parallel execution on multi-core machines [28].

## 7 Conclusion

Diversity defenses are a promising mechanisms for making vulnerable servers more difficult for attackers to exploit. The effectiveness of a diversity defense depends on what properties of the execution it alters, the amount of entropy in the randomization, and how resistent the diversity defense is to attempts to probe the system or to circumvent or deputize the diversity mechanisms. Dynamic diversity can enhance the effectiveness of these defenses by rerandomizing the system periodically or based on trigger events. The effectiveness of dynamic diversity, however, is limited to scenarios where the attack requires an extended sequence of requests to probe the system and develop an attack. For many scenarios, dynamic rerandomization provides less benefit than expected. It provides no benefit against circumvention and deputy attacks, and against entropy reduction attacks provides at most a factor of two increase in difficulty. Against other classes of attack, dynamic diversity defense may provide more substantial advantages, but the defenses must be crafted carefully to provide the intended benefits.

## Acknowledgment

This research has been partially supported by grants from the National Science Foundation and the US Department of Defense under AFOSR MURI grant FA9550-07-1-0532. 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 sponsoring agencies.

## References


1. Alexander Peslyak (Solar Designer). Return-to-libc Attack. Bugtraq Mailing List, August 1997.
2. Emery D. Berger and Benjamin G. Zorn. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2006.
3. Sandeep Bhatkar, Daniel DuVarney, and R. Sekar. Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *USENIX Security Symposium*, 2003.
4. Stephen W. Boyd, Gaurav S. Kc, Michael E. Locasto, Angelos D. Keromytis, and Vassilis Prevelakis. On The General Applicability of Instruction-Set Randomization. *IEEE Transactions on Dependable and Secure Computing*, 7(3), 2010.
5. Kevin Brown. Balls In Bins with Limited Capacity. http://www.mathpages.com/home/kmath337.htm.
6. Brian X. Chen. Apple's Snow Leopard Is Less Secure Than Windows, But Safer. *Wired*, September 2009.
7. Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-Control-Data Attacks Are Realistic Threats. In *USENIX Security Symposium*, 2005.
8. Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointGuard: Protecting Pointers from Buffer Overflow Vulnerabilities. In *12th USENIX Security Symposium*, 2003.
9. Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. N-Variant Systems: A Secretless Framework for Security through Diversity. In *USENIX Security Symposium*, 2006.
10. Cristian Cadar and Periklis Akritidis and Manuel Costa and Jean-Phillipe Martin and Miguel Castro. Data Randomization. Technical Report TR-120-2008, Microsoft Research, 2008.
11. Tyler Durden. Bypassing PaX ASLR protection. http://www.phrack.com/issues.html?issue=59&id=9/, 2009.
12. Elena Gabriela Barrantes and David Ackley and Stephanie Forrest and Trek Palmer and Darko Stefanovic and Dino Dai Zovi. Intrusion Detection: Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. In *10th ACM Conference on Computer and Communications Security (CCS)*, 2003.
13. Elena Gabriela Barrantes and David H. Ackley and Stephanie Forrest and Darko Stefanovic. Randomized Instruction Set Emulation. *ACM Transactions on Information and System Security*, February 2005.
14. Gaurav S. Kc and Angelos D. Keromytis and Vassilis Prevelakis. Countering Code-Injection Attacks with Instruction-Set Randomization. In *10th ACM Conference on Computer and Communications Security (CCS)*, 2003.
15. Sudhakar Govindavajhala and Andrew W. Appel. Using Memory Errors to Attack a Virtual Machine. In *IEEE Symposium on Security and Privacy (Oakland)*, 2003.
16. Norman Hardy. The Confused Deputy (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22(4), October 1988.
17. David Holland, Ada Lim, and Margo Seltzer. An Architecture A Day Keeps The Hacker Away. In *Workshop on Architectural Support for Security and Anti-Virus*, April 2004.
18. Kubuntu Wiki. Supported Position Independent Executables. https://wiki.kubuntu.org/SecurityTeam/KnowledgeBase/BuiltPIE, 2011.
19. Microsoft Corporation. Microsoft Security Advisory (961051): Vulnerability in Internet Explorer Could Allow Remote Code Execution. http://www.microsoft.com/technet/security/advisory/961051.mspx, December 2008.
20. Tilo Müller. ASLR Smack and Laugh Reference. Seminar on Advanced Exploitation Techniques, February 2008.
21. Ryan Naraine. Adobe PDF Exploits Using Signed Certificates, Bypasses ASLR/DEP. *ZDNet Zero Day*, September 2010.

22. Anh Nguyen-Tuong, David Evans, John C. Knight, Benjamin Cox, and Jack W. Davidson. Security through Redundant Data Diversity. In *IEEE/IFPF International Conference on Dependable Systems and Networks*, June 2008.
23. Anh Nguyen-Tuong, Andrew Wang, Jason D. Hiser, John C. Knight, and Jack W. Davidson. On the effectiveness of the metamorphic shield. In *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, ECSA '10, pages 170–174, New York, NY, USA, 2010. ACM.
24. Pratap V. Prahbu and Yingbo Song and Salvatore J. Stolfo. Smashing the Stack with Hydra: The Many Heads of Advanced Polymorphic Shellcode. Technical Report CUCS-037-09, Columbia University, August 2009.
25. Rapid7 LLC. Metasploit. http://www.metasploit.com/, 2003–2011.
26. Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin Zorn. Nozzle: A Defense Against Heap-spraying Code Injection Attacks. In *USENIX Security Symposium*, 2009.
27. Babak Salamat, Andreas Gal, and Michael Franz. Reverse Stack Execution in a Multi-Variant Execution Environment. In *Workshop on Compiler and Architectural Techniques for Application Reliability and Security*, June 2008.
28. Babak Salamat, Todd Jackson, Andreas Gal, and Michael Franz. Orchestra: Intrusion Detection using Parallel Execution and Monitoring of Program Variants in User-Space. In *ACM European Conference on Computer Systems (EuroSys)*, 2009.
29. Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *ACM Conference on Computer and Communications Security (CCS)*, CCS '04, pages 298–307, New York, NY, USA, 2004. ACM.
30. Alexander Sotirov. Heap Feng Shui in JavaScript. http://www.blackhat.com/presentations/bh-europe-07/Sotirov/Presentation/bh-eu-07-sotirov-apr19.pdf, 2007.
31. Ana Nora Sovarel, David Evans, and Nathanael Paul. Where's the feeb? the effectiveness of instruction set randomization. In *14th USENIX Security Symposium*, Berkeley, CA, USA, 2005. USENIX Association.
32. Stephanie Forrest and Anil Somayaji and David Ackley. Building Diverse Computer Systems. In *Hot Topics in Operating Systems*, 1997.
33. Stephen W. Boyd and Angelos D. Keromytis. SQLrand: Preventing SQL Injection Attacks. In *Applied Cryptography and Network Security (ACNS)*, 2004.
34. Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund, and Thomas Walter. Breaking the Memory Secrecy Assumption. In *Second European Workshop on System Security*, 2009.
35. PaX Team. PaX Homepage. http://pax.grsecurity.net/, 2000.
36. Wei Hu and Jason Hiser and Dan Williams and Adrian Filipi and Jack W. Davidson and David Evans and John C. Knight and Anh Nguyen-Tuong and Jonathan Rowanhill. Secure and Practical Defense Against Code-injection Attacks Using Software Dynamic Translation. In *Second International Conference on Virtual Execution Environments*, 2006.
37. Yoav Weiss and Elena Gabriela Barrantes. Known/Chosen Key Attacks against Software Instruction Set Randomization. In *Annual Computer Security Applications Conference (ACSAC)*, 2006.
38. Berend-Jan "SkyLined" Wever. MS Internet Explorer (IFRAME Tag) Buffer Overflow Exploit. http://www.exploit-db.com/exploits/612/, 2004.