Composition Challenges for Automated Software Diversity

Benjamin Davis
Galois, Inc.
ben@galois.com

Per Larsen
Immunant, Inc.
perl@immunant.com

Stijn Volckaert
University of California, Irvine
stijnv@uci.edu

Simon Winwood
Galois, Inc.
sjw@galois.com

David Melski
GrammaTech, Inc.
melski@grammatech.com

Michael Franz
University of California, Irvine
franz@uci.edu

Stephen Magill
Galois, Inc.
stephen@galois.com

ABSTRACT

Over the past 20 years, a variety of automated software diversity techniques have been proposed. Some techniques randomize aspects of the implementation that are left undefined by the source language specification, such as code layout, stack layout, or locations of heap-allocated objects. Others insert instrumentation or obfuscation that is transparent from an application perspective, e.g. using XOR masks to obscure data values in memory or hiding code pointers using jump tables. A common assumption is that layering these techniques improves security due to increased entropy in the resulting binary. In this paper we examine this assumption and show that it fails to hold in general. In particular, it fails in one of the strongest deployment models for software diversity—that of multiple diverse variants running together in a multi-variant execution environment (MVEE) where attacks manifest as detectable behavioral divergence. We present several examples of diversity combinations that are vulnerable to attack in an MVEE even when none of the component techniques are vulnerable in isolation. Based on these results, we present guidance on which techniques do combine well and suggestions for effective deployment of diversity in MVEEs.

CCS Concepts

•**Security and privacy** → *Vulnerability management; Software security engineering; Intrusion/anomaly detection and malware mitigation;*

Keywords

multi-variant execution environments; composing defenses; attack detection

1. INTRODUCTION

LAW 2016 Los Angeles, California

DOI: 10.1145/1235

A large portion of currently-deployed and newly-developed applications are written in unsafe languages such as C and C++. The memory safety violations that are possible in such languages enable many of the attacks that now occur on a regular basis.

A variety of mitigations have been proposed for these problems; we distinguish between *enforcement* techniques and those based on *randomization*. Enforcement is concerned with preventing memory safety violations such as out-of-bounds memory accesses [19], preventing control-flow hijacking [1], or protecting code pointers [15]. These techniques can provide absolute assurance against certain attack techniques, but at the cost of increased execution overhead.

Randomization, often referred to as *software diversity*, instead varies aspects of the code that are implementation-defined, such as stack variable orderings or code layout. Since there are many choices for these aspects of the implementation, randomizing compilers can easily generate many program *variants*, all of which share any latent memory safety weaknesses, but potentially differ in the degree of exploitability of those weaknesses (or perhaps even whether exploitation is possible). An advantage of randomization is that these changes typically have no significant performance impact. A disadvantage is that the security guarantees are probabilistic.

Multi-variant execution environments (MVEEs) run multiple variants of a program on the same inputs while monitoring for differences in behavior. Exploitation of a program in an MVEE thus requires eventual exploitation of all variants. At first glance such a system appears to clearly increase the security offered over what can be achieved with randomization alone (e.g. a single randomized variant). It might seem reasonable to assume that if N variants must be exploited, and each variant has some probability P of being exploitable, then perhaps the probabilities multiply and the N-variant system has probability P^N of being exploitable.

In this paper, we show that such an increase in security is not necessarily guaranteed. In fact, careful choice of randomization techniques is required to realize the benefit of MVEEs. Our fundamental result is that *diversity does not compose in general*. There are two axes along which composition results may be considered.

1. **Composition of diversity techniques** Is the combination of technique A and technique B more secure than either in isolation?

2. **Composition of variants** Is running four variants in an MVEE more secure than running two? Is running two more secure than running a single variant in isolation?

Our finding is that neither type of composition provides for increased security in general. We demonstrate this by describing several attacks on variants running in an MVEE. We then provide suggestions for combinations of techniques that do work well together in a multi-variant context and minimize susceptibility to these attacks.

2. BACKGROUND

In the physical domain, we use a combination of obstacles, hiding, and detection to counter unwanted behavior such as theft. In the home, for example, we lock our front door, hide our valuables, and use guards and sensors to detect intruders. In the digital domain, we prevent exploitation of vulnerable programs using a similar combination of obstacles, hiding, and detection. Obstacles include $X \oplus W$ policies which prevent runtime code injection and code signatures which protect the integrity of programs on disk. We hide the internals of potentially vulnerable programs (§ 2.1) and use MVEE systems to add detection capabilities (§ 2.2).

2.1 Diversity

The compiler can translate source code to machine code in a number of ways as long as it carries out the high level operations specified by the programmer. Exploit payloads, on the other hand, are so dependent on program internals that even changing compiler flags can render the payload inert. Artificial software diversity takes this idea one step further by introducing new code transformations whose sole purpose is to randomize the in-memory program representation, effectively hiding the attack surface from adversaries [16].

Diversity transformations target particular implementation aspects. If, for example, an exploit makes assumptions about the layout of stack frames, defenders may randomize the orders of buffers and scalars on the stack. Simiarly, exploits including a Return-Oriented Programming (ROP) [24] stage become unreliable when diversity transformations are used to randomize the code layout. Because deployment of diversity techniques shifts the cost and feasibility of various exploitation techniques, it is necessary to use multiple diversity techniques (and integrity techniques) to raise the bar to exploitation across the board.

At the highest level, diversity transformations target either the code or the data layout of the program representation. Another key property is the granularity. Address Space Layout Randomization (ASLR) is at the coarse end of the spectrum because it randomizes the base address of the `.text` section as well as the heap and stacks but doesn't randomize anything inside each section [21]. Medium-grain code diversity techniques include those that randomly shuffle code pages or functions (Figure 2) whereas the most fine-grained techniques shuffle basic blocks or randomize individual instructions [10, 8, 29].

Data-oriented diversity transformations target the stack, global and heap layouts or finer-grainer properties such as

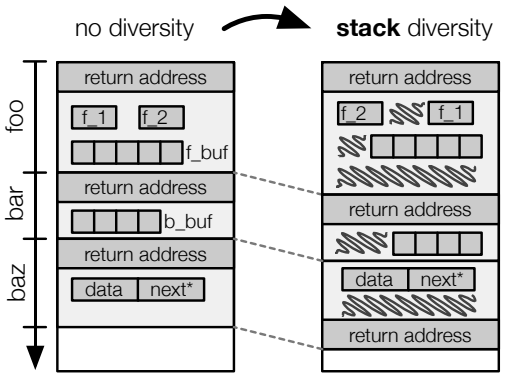

Figure 1: Shuffling and padding stack frames.

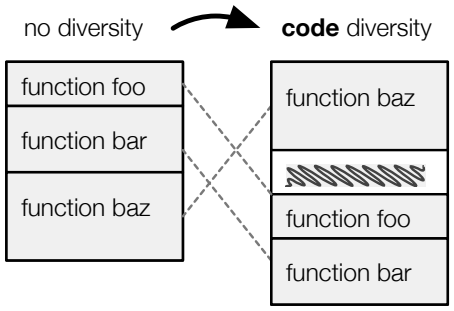

Figure 2: Shuffling and padding the code layout.

the layout of individual allocations or even the representation of data itself. The order in which variables are assigned stack frame slots, for instance, can be randomized, and padding variables can be introduced to increase the number of possible layouts (Figure 1). Similar techniques can be used to randomize the layout of global variables.

The layout of the heap is chosen by the heap allocator at run time, rather than by the compiler. This makes it natural to randomize and pad heap objects dynamically as they are allocated. To target use-after-free exploitation techniques, heap diversifiers also delay and randomize memory deallocation.

Cross-cutting, data-oriented transformations include data randomization and structure layout randomization. Data randomization i) uses static program analysis to divide all legitimate data flows into a number of equivalence classes and ii) assigns a random bitmask to each load and store operation according to its equivalence class [3, 2]. This way, unintended, malicous data flows that cross equivalence classes will be corrupted due to masking, while intended data flows remain unaffected (see Figure 3). Structure layout randomization, as the name implies, shuffles and pads fields within (non-packed) structures and classes [5, 17].

2.2 Multi-Variant Execution Environments

MVEEs execute two or more diversified programs (variants) in lockstep while monitoring their behavior at the level of system calls. Execution is terminated if the MVEE detects divergence. Because each diversified variant receives the same program input but responds differently to memory

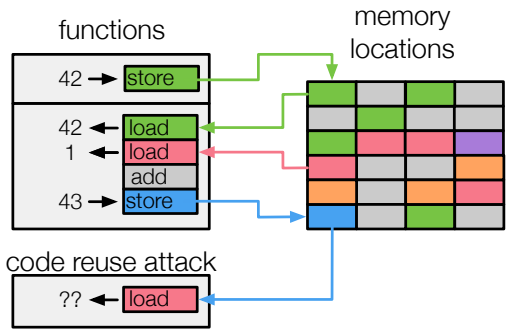

Figure 3: With data randomization, each load and store instruction applies a bitmask to data values. Legitimate loads and stores to the same data object uses compatible bitmasks whereas malicious accesses are likely mismatched thus garbling the data.

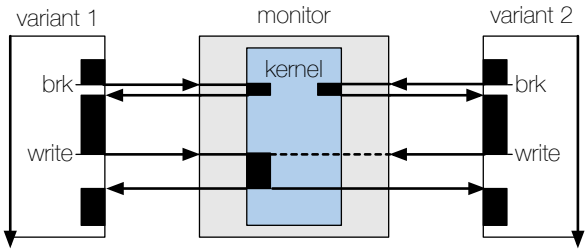

Figure 4: Monitoring and replication between two variants.

corruption, adversaries must simultaneously and reliably exploit N program variants without causing them to diverge.

The MVEE must synchronize the variants and present them as a single application to the end-user. To do so, the MVEE monitor duplicates program inputs once for each variant; variant outputs are similarly compared and deduplicated such that each output operation is performed only once. MVEEs monitor variants by interposing on the system calls made by each variant. In other words, the monitor gains control over a variant each time it makes a system call. This lets the monitor decide whether to forward the system call to the kernel and decide if and when variant execution is resumed (Figure 4).

A variety of MVEE designs were explored in the past decade. Although these designs have much in common, there are some interesting features that distinguish them.

The majority of existing designs use a single, centralized, monitor component that is placed outside the address spaces of the variants. Some of these designs have a monitor that runs in kernel space [7, 6], whereas others run the monitor as a standalone user-space process that attaches to the variants using the operating system's debugging APIs [22, 28, 4, 18, 11].

More recently, several researchers proposed decentralized designs, where a seperate monitor is used for each variant [12, 27, 14]. This monitor can then be placed into the variants' address spaces. Decentralized monitors communicate through a shared memory region that is set up when the monitors start. This shared memory region contains a

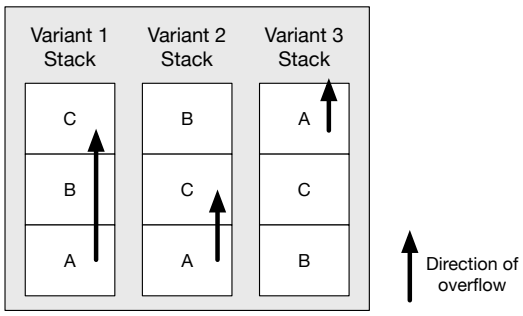

Figure 5: Stack layout in three example variants.

ring buffer that is optimized to hold system call arguments and return values.

At least two of the existing MVEEs offer the ability to set up additional ring buffers that can be used to perform additional divergence detection below the system call interface [6, 27]. These ring buffers can be used to compare function call arguments across variants, for example.

3. MVEE DIVERSITY ATTACKS

In order to exploit a set of variants running in an MVEE, it is first necessary that each variant be vulnerable in isolation. For example, suppose we have the situation in Figure 5, in which we have three integer-valued stack variables: A, B, and C. Suppose that the attacker can overflow A and that the variant is successfully exploited if the attacker can write 0x3f into C (perhaps C represents a permission or authorization level). In variants 1 and 2, C follows A on the stack and so this overwrite can occur. In variant 3, C is below A on the stack and so the attacker cannot achieve the win condition by overflowing A. We say that the vulnerability is not exploitable in variant 3, and by extension any variant set including variant 3 will not be exploitable.

In a variant set including variants 1 and 2, the attacker has two options for exploiting the vulnerability.

1. **Parallel Exploitation** He can overflow from A to C in both variants simultaneously. This requires overflowing A by two words, each containing 0x3f. This will overwrite both B and C with 0x3f and will only work if storing 0x3f in B does not interfere with program behavior (perhaps B is not referenced again or is overwritten later with uncorrupted data).

2. **Serial Exploitation** He can perform a single-word overflow, which overwrites C in variant 2 while leaving B uncorrupted. If successful attack requires not altering B, then the attacker can exploit variant 2 and then launch the attack separately against variant 1.

We will consider both types of attack in this section.

3.1 Attack Primitives and Attacker Goals

The effect of diversity techniques on the ability of an attacker to achieve his goal depends on the *mechanism* the attacker is using to accomplish some effect. In this section we detail both the various attack primitives we consider and the attacker goals that we attempt to prevent via diversity.

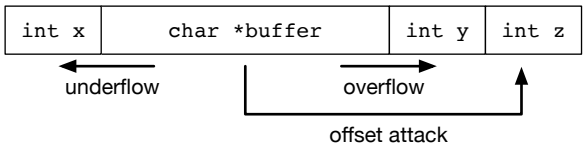

Figure 6: Attack primitives we consider.

3.1.1 Attack Primitives

Figure 6 provides a graphical reference to our attack primitives, which we describe in more detail below.

Buffer Overflow In this form of attack, the adversary is able to write data past the end of a buffer, overflowing data stored at higher addresses.

Buffer Underflow The adversary is able to write data that gets stored at memory addresses prior to the first address that is part of the buffer.

Offset Attack The adversary is able to write an arbitrary value at an arbitrary offset from some program object.

Attacks of the above three types are often *bounded*. For example, an attacker may only be able to overflow a buffer by 4 bytes. Or offset attacks may be limited to word-sized values. Word-sized offset vulnerabilities are particularly useful from an attacker point-of-view, since they provide a very flexible and precise primitive that minimizes unintended data corruption.

3.1.2 Win Conditions

The state changes that may enable attackers to achieve their goals are varied and numerous. In order to focus our investigations, we have concentrated our effort on a few types of state change that are known to play a part in attacks.

Chosen Value In this case, an attacker "wins" if he is able to write a chosen value to a chosen object. For example, "write the value 5 to `creds->auth_level`."

Chosen Function Success in this case requires writing the address of an attacker-chosen function to a particular object. For example, "write `&grant_access` to `callbacks->handle_auth_failure`".

Data Property In some cases, the value an attacker writes merely has to satisfy some property to result in attack success. For example, suppose there is a 32-bit field `creds->is_admin` that is used as a flag. An attack on such a value may succeed provided the attack can write any non-zero value to `creds->is_admin`.

In many cases, real attacks will face additional constraints. For example, perhaps an attacker has to write a value that is both a function address and a valid ASCII character string. However, if we can show that chosen value attacks are defended, then this necessarily defends against attacks that have various constraints on the value written.

3.2 Attack Examples

We now discuss example attacks that are feasible against variants running in an MVEE when certain combinations of diversity techniques are applied. Critically, in many cases these techniques provide more security in isolation than they do when they are composed. As such, these serve as counterexamples to general composability of diversity techniques in an MVEE context.

Additionally, parallel exploitability of a variant collection in an MVEE corresponds to attack generality in a non-MVEE context. In a non-MVEE context, the allure of software diversity is that it greatly increases the attacker work factor. Instead of constructing a single payload that can exploit any instance of an application, the attack must construct different payloads for each application instance. However, if it is possible to construct an attack that simultaneously exploits a collection of variants in an MVEE context, it is also possible to construct a single payload that exploits all of those variants in isolation (e.g. in a non-MVEE deployment scenario).

3.2.1 Skewed Return Attack

Suppose we have the setup depicted in Figure 7. At the left of the figure we show the call stack for the program. We assume that an access of object A can overflow, giving the attacker the ability to write past the end of the stack slot holding A. The attacker's goal is to overwrite the return address, which currently points to Proc1 (solid line), with a reference to Proc2 (dashed line). Such an attack captures a "return to `libc`" exploitation scenario. We present three variants, where variants 1 and 2 differ in their code layout (Proc1 is at a different address in each variant, as is Proc2). Variant 3 is the result of combining code layout diversity and stack layout diversity, where the order of items on the call stack is shuffled. Thus variants 1 and 3 differ in both their code layout and their stack layout.

Variants such as Variant 1 and Variant 2, that have had their code layout randomized, provide strong protection against stack-based control-flow hijack attacks, as depicted in Figure 8. Scenario A in that figure shows what the attacker would like to write to the stack in each variant, labeled *Goal 1* and *Goal 2*. As we can see, since Proc2 is at a different address in each variant, and the return address is three stack slots away from object A in each variant, these goals require that different values be written into the slot three positions up from object A. Since there is only a single payload, this slot can only contain a single value. If we assume that code is mapped at disjoint addresses in each variant, then it is impossible for the attacker to generate a payload that stores the required address in each variant.

Scenario B in Figure 8 illustrates how adding an additional diversity technique can actually weaken the level of protection offered. Despite variant 3 adding stack layout diversity, the combination of variants 1 and 3 in an MVEE is simultaneously exploitable, as the variants now have different relative offsets between the source object A and the target slot (the return address). Thus, a single payload can be constructed that writes different values to the return address in each variant, acheiving parallel exploitation of the variants in the MVEE.

In the terminology of Section 3.1.2, by adding stack layout permutation, we have enabled chosen function attacks. A similar effect occurs with data randomization. Just as

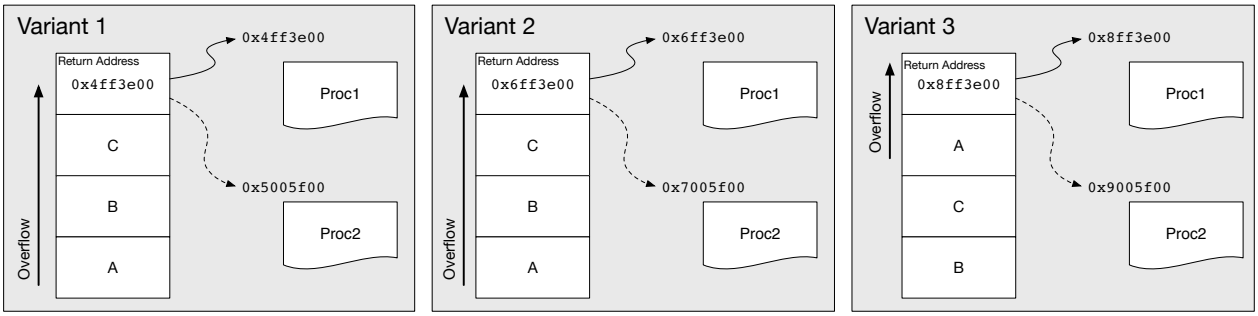

Figure 7: Three variants demonstrating the skewed return attack.

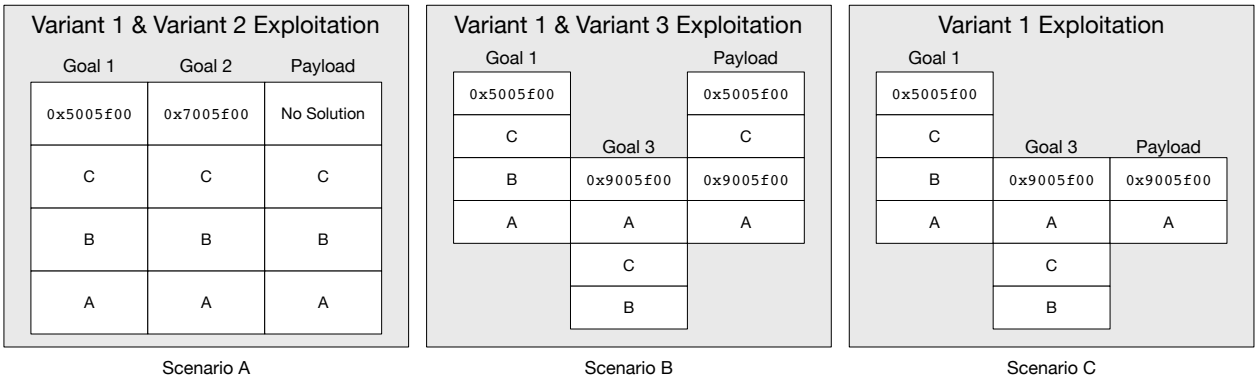

Figure 8: An example of a variant set immune from attack (Scenario A) and one where adding stack layout randomization enables a control-flow hijack attack (Scenario B).

code layout randomization with a fixed stack layout has the potential to completely block chosen function attacks, so data randomization can prevent chosen value attacks. Introducing stack layout diversity then enables a data-only variant of the skewed return attack. Furthermore, the same phenomenon, where layout changes enable new attack vectors, occurs in the global space and on the heap as well. Thus, when holding layouts constant it is important to do this across all of the program's segments. We further discuss defenses against Skewed Return style attacks in Section 4.

3.2.2 Skewed Data

An additional sort of data-only attack is also feasible against variant sets protected with data layout diversity. While data layout diversity provides strong protection against chosen value attacks, the attacker needn't always need full control over the value that is written to accomplish the intended effect. In cases such as integers used as boolean flags, any value other than zero will be interpreted as "true". Thus, if an attacker wishes to flip a branch that will evaluate to "false" under non-attack conditions, simply writing a random value is highly likely to result in the intended effect.

Data randomization divides the program's data into sets and associates different masks with each set. Values are XORed with these masks when they are written to memory, thus obscuring the true memory contents. On memory reads, they are once again XORed with the mask, restoring the original value. If we take the variable sets to be static alias classes, then a write that crosses alias classes—that is, a memory error—will store a value that is essentially random (it is the attacker's value XORed with a randomly chosen mask). However, as we just detailed above, writing random values may be sufficient to gain control in some cases.

Furthermore, the problem extends beyond Boolean flags. Buffer lengths stored in memory can be overwritten with random values that will, with high probability, increase these bounds. An increased stored length value will no longer properly protect the associated buffer, enabling further memory corruption attacks. We call these attacks *skewed data* attacks, since they are enabled whenever the set of safe data values is much smaller than the set of unsafe data values.

Skewed data attacks can potentially succeed at establishing the Data Property win condition from Section 3.1.2. The cross checks described in Section 2.2 can prevent these attacks, and we detail this protection in Section 4.

3.2.3 Padding and Isolated Attacks

Inside of an MVEE, an attack may result in writing some fixed value to different, non-corresponding regions of memory in each variant. Not only is the impact of an attack dependent on what is overwritten, but this can also constrain an attacker's capabilities. For example, different layouts may force an overflow that overwrites a `username` variable in one variant to overwrite a function pointer in another. Without further manipulations, the attacker cannot write a username that is an invalid function pointer without being detected when the function pointer is used.

While padding can increase the number of ways a data layout may be shuffled, inserting padding into variants can help the attacker when it removes constraints on an attacker's payload. If the attacker can overwrite a variable of interest in a single variant, but only unchecked padding in the others, then they can attack that variant independently of the others. Alternatively, attackers may be able to overtake the entire set by attacking each variant individually, such as by overwriting function pointers in each variant in turn before the pointer is accessed by the running programs. It is better to constrain the possible attacks by using canaries or cross-checked values in lieu of unchecked padding.

Consider the system depicted in Figure 9, which may be considered analogous to that shown in Figure 7, except that rather than having three variables, there is only one, and so padding has been introduced. Similarly to the attack shown in Figure 8, a successful attack is possible by taking advantage of the skewed nature of the stack; in contrast to Figure 8, however, the attacker-inserted return address overwrites unused padding in the first variant rather than a potentially live variable. Thus, padding has further eased the attacker's burden.

3.2.4 Cross-stack-frame attacks and SafeStack

Some existing stack protection techniques like SafeStack [15] limit stack buffer overflows by moving stack variables potentially vulnerable to overflow onto a separate "unsafe stack". Return addresses and other variables only accessed safely are placed on the "safe stack," and the attacker cannot overflow from the unsafe stack onto these values.

Buffer overflows that write past the end of a stack variable and continue on to overwrite a variable in a parent stack frame can be very difficult to exploit in an MVEE setting. When code layout has been randomized, it can be difficult to overwrite the saved return address with desirable values in all variants. Furthermore, overwriting the saved base pointer addresses stored on each variant's stack can be difficult when the desired values are in disjoint regions from variant to variant, such as when applying ASLR. However, applying protection techniques like SafeStack can make exploiting these overflows that span stack frames easier. After applying SafeStack, elements on the unsafe stack are adjacent to unsafe stack elements from the parent stack frame, but the saved base pointers and return addresses no longer exist between the two, as they are on the safe stack. Now the attacker may be able to perform data-only overwrites from the unsafe stack onto variables on the parent stack frame's unsafe stack without being constrained by writing valid base pointer and return address values in all variants.

4. SECURE VARIANT SETS

As a byproduct of exploring the MVEE attack space, we have developed some best practices in terms of variant selection for MVEEs. The approach is based around combining *randomized data* with *deterministic layouts*. Since MVEEs attempt to run variants in parallel as much as possible, and modern CPUs frequently have at least four cores, we present suggestions for sets of four variants that together avoid the attacks described in Section 3.2. The variants accomplish this while still providing for some layout randomization to help protect against attack classes other than those considered here.

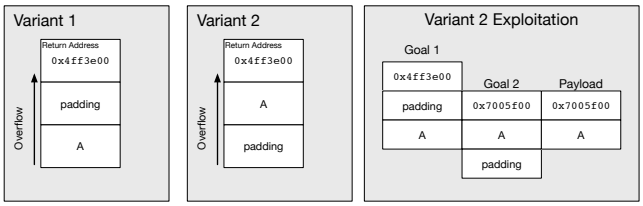

Figure 9: Two variants demonstrating padding-induced vulnerabilities.

Figure 10 presents a graphical overview of our variant set construction. The color of the main enclosing box indicates data layout, with matching colors indicating matching layouts. So variant 1 and 2 have the same relative layout for the stack, the globals, and the heap, a property we call *twinning*. Section-level randomization such as ASLR may relocate the stack base, heap base, and global section, but within each section the relative positioning of objects will be the same. This ensures that relative overflows from object A with offset o within a single memory section (stack, heap, or globals) will necessarily land in the same target object.

The color of the internal boxes represent code layout diversity and data randomization keys, with matching colors indicating matching layout and matching keys. Crucial to the security of this combination of variants is the fact that varying code layout and data randomization keys changes the expected contents of memory. So the value of a pointer to function `f` will be different in different variants due to code layout randomization. A data value stored in a global will be different due to data randomization. When these per-object variations are combined with the consistent relative positioning of objects that twinning provides, it helps to set up the situation in Scenario B of Figure 8, where certain types of exploitation are not possible due to the attacker's inability to generate a payload that has the intended effect on each variant.

The security properties described so far can be provided by just variant 1 and 2 together. Adding variants 3 and 4, which follow the same construction methodology but with new relative layouts for the stack, heap, and globals, helps to provide protection of the sort depicted in variant 3 from Figure 5, where the source of the overflow (A) and the target (C) are positioned such that the target cannot be overwritten. If more parallelism is available, even more pairs of twinned variants could be included, increasing the chances that at least one set makes a particular targeted relative write impossible.

Finally, in order to protect against the *skewed data* attack described in Section 3.2.2, we can include cross checks that compare the un-masked data values in each variant to ensure that they match. With data randomization enabled, a write into an object A that crosses alias set boundaries (e.g. a write resulting from a memory safety vulnerability) will result in different values being written to each cell. While it may be sufficient to write a non-zero value in order to flip a conditional branch and thus alter program execution, each variant will contain a *different* non-zero value, which the cross check will detect. Our variant generation tools currently support inserting such custom cross-checks for all values that are read from memory and flow into conditional branches.

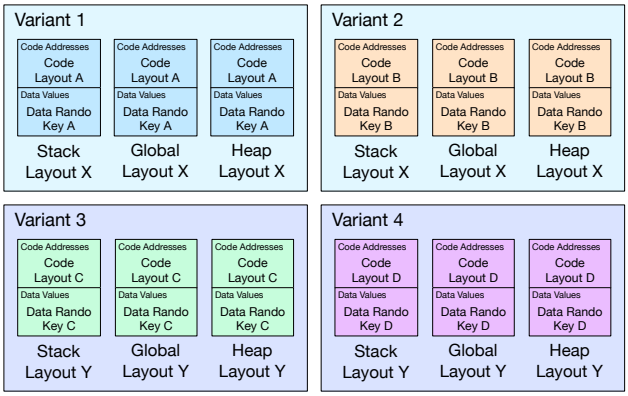

Figure 10: Our suggested combination of diversity techniques for maximizing diversity-based protection while avoiding the attacks described in Section 3.2.

Limitations.

The combination of techniques described here are selected to avoid the attacks described in Section 3.2 and maximize the changes of detecting attempted exploitation. However, because many elements of variant generation are randomized, there is still a small probability of failure. For example, data randomization could choose keys that are the same across variant sets (though this would be highly unlikely), and stack layout randomization could choose unfortunately stack layouts that place some objects at the same relative positions across variants.

Heap randomization adds additional entropy challenges, since overall entropy level has to be balanced with practical concerns such as memory fragmentation and page table size. We are currently exploring methods to both minimize the chances of unintended cross-variant overlap as well as to deterministically generate variant sets with good security characteristics.

5. RELATED WORK

There has been prior work on the security of diversity techniques in a single-variant context [9, 23, 25, 26]. In contrast to this work, we focus on attacks on diversity in a multi-variant context. Common attack avenues used in prior work such as timing channels or information disclosure vulnerabilities seem difficult or impossible to exploit in a multi-variant context. In our work, we show that there are attacks on diversity that succeed in a multi-variant context. These attacks rely on a stronger attacker model, which includes prior knowledge of implementation details of the variants. We consider this to be a fair assumption since it is the same threat model targeted by multi-variant execution work such as [7].

Cox et al. [7] proposed N-variant systems as a security technique in 2006 and noted that security depends on having, for each attack class targeted, a pair of variants that include differences designed to block that class. For example, their address space partitioning technique is designed to disrupt attacks that depend on writing absolute code addresses. They note the pitfalls with respect to arbitrary composition but do not give specific examples of attacks, which we provide here. They also provide a method for combining n diversity techniques safely using $n + 1$ variants, while speculating that more efficient combinations may be possible without sacrificing security. In this paper we provide such an efficient combination, combining 5 diversity techniques into 4 variants, although with probabilistic rather than absolute guarantees in some cases.

Techniques similar to our data randomization techniques were considered in [20]. These techniques rely on leveraging domain knowledge to change the mapping from concepts to representation. The data randomization technique we apply is fully automated and only uses knowledge that can be obtained through static program analysis of aliasing relationships. This provides probabilistic rather than absolute security guarantees.

In [13], probabilistic diversity techniques such as those we consider are experimentally compared in a multi-variant execution environment. Good combinations of diversity techniques were identified based on these experiments. One shortcoming, noted in the paper, is that simple recompilation often breaks concrete attacks and the question of whether an attack could be tweaked or adapted to work in a multi-variant environment is difficult to answer at scale. This limits the extent to which large-scale experimentation can be used to evaluate diversity effectiveness. In this work we consider general attack primitives rather than specific attacks, sidestepping this issue.

6. CONCLUSION AND FUTURE WORK

In this paper we have provided an initial exploration of the attack space for multi-variant systems. We have described attacks that are possible in such systems and demonstrated that adding additional diversity techniques can actually compromise security in a multi-variant context. We also presented a method for constructing variant sets that avoid these attacks, while also incorporating randomness across variants for probabilistic protection against additional attacks.

Together these results enable more effective use of multi-variant execution environments. Future avenues of exploration include considering additional attack classes and diversity techniques. Additionally, the security guarantees provided by some techniques are still probabilistic in nature. For example, stack permutation only probabilistically prevents stack overflows from reaching target variables. Going forward, we will investigate whether these security guarantees can be made to hold in all cases.

Acknowledgments

This material is based on work supported by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under contract number FA8750-15-C-0124. Opinions expressed herein are our own.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Defense Advanced Research Projects Agency, its Contracting Agents, the Air Force Research Laboratory, or any other agency of the U.S. Government.

7. REFERENCES

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM*

conference on Computer and Communications Security, CCS '05, pages 340–353, 2005.

[2] S. Bhatkar and R. Sekar. Data space randomization. In *Proceedings of the Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '08, pages 1–22, 2008.

[3] C. Cadar, P. Akritidis, M. Costa, J.-P. Martin, and M. Castro. Data randomization. Technical Report MSR-TR-2008-120, Microsoft Research, September 2008.

[4] L. Cavallaro. *Comprehensive Memory Error Protection via Diversity and Taint-Tracking*. PhD thesis, Univ. Degli Studi Di Milano, 2007.

[5] P. Chen, J. Xu, Z. Lin, D. Xu, B. Mao, and P. Liu. A practical approach for adaptive data structure layout randomization. In *Computer Security - ESORICS 2015*, volume 9326 of *Lecture Notes in Computer Science*, pages 69–89. Springer International Publishing, 2015.

[6] M. Co, J. W. Davidson, J. D. Hiser, J. C. Knight, A. Nguyen-Tuong, W. Weimer, J. Burket, G. L. Frazier, T. M. Frazier, B. Dutertre, et al. Double helix and raven: A system for cyber fault tolerance and recovery. In *Proceedings of the 11th Annual Cyber and Information Security Research Conference*. ACM, 2016.

[7] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser. N-variant systems: A secretless framework for security through diversity. In *USENIX Security Symposium*, pages 105–120, 2006.

[8] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson. ILR: Where'd my gadgets go? In *Proc. of the 33rd IEEE Symposium on Security and Privacy*, S&P '12, pages 571–585, 2012.

[9] T. Hobson, H. Okhravi, D. Bigelow, R. Rudd, and W. Streilein. On the challenges of effective movement. In *Proceedings of the First ACM Workshop on Moving Target Defense*, MTD '14, pages 41–50, New York, NY, USA, 2014. ACM.

[10] A. Homescu, T. Jackson, S. Crane, S. Brunthaler, P. Larsen, and M. Franz. Large-scale automated software diversity—program evolution redux. *IEEE Transactions on Dependable and Secure Computing*, 2015.

[11] P. Hosek and C. Cadar. Safe software updates via multi-version execution. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 612–621, 2013.

[12] P. Hosek and C. Cadar. VARAN the unbelievable: An efficient n-version execution framework. In *Proc. Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 339–353, 2015.

[13] T. Jackson, B. Salamat, G. Wagner, C. Wimmer, and M. Franz. On the effectiveness of multi-variant program execution for vulnerability detection and prevention. In *Proceedings of the 6th International Workshop on Security Measurements and Metrics*, MetriSec '10, pages 7:1–7:8, New York, NY, USA, 2010. ACM.

[14] K. Koning, H. Bos, and C. Giuffrida. Secure and efficient multi-variant execution using hardware-assisted process virtualization. In *IEEE/IFIP International Conference on Dependable Systems and Networks*, 2016.

[15] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[16] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. SoK: Automated software diversity. S&P '14, 2014.

[17] Z. Lin, R. Riley, and D. Xu. Polymorphing software by randomizing data structure layout. In *Detection of Intrusions and Malware, and Vulnerability Assessment, 6th International Conference*, 2009.

[18] M. Maurer and D. Brumley. Tachyon: Tandem execution for efficient live patch testing. In *USENIX Security Symposium*, pages 617–630, 2012.

[19] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Softbound: highly compatible and complete spatial memory safety for c. In *ACM Sigplan Notices*, volume 44, pages 245–258, 2009.

[20] A. Nguyen-Tuong, D. Evans, J. C. Knight, B. Cox, and J. W. Davidson. Security through redundant data diversity. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pages 187–196, June 2008.

[21] PaX Team. Address space layout randomization. http://pax.grsecurity.net/docs/aslr.txt, 2004.

[22] B. Salamat, T. Jackson, A. Gal, and M. Franz. Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space. In *Proc. EuroSys Conf.*, pages 33–46, 2009.

[23] J. Seibert, H. Okhravi, and E. Söderström. Information leaks without memory disclosures: Remote side channel attacks on diversified code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 54–65, New York, NY, USA, 2014. ACM.

[24] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proc. ACM Conf. Computer and Communications Security*, pages 552–561, 2007.

[25] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, pages 298–307, New York, NY, USA, 2004. ACM.

[26] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 574–588, Washington, DC, USA, 2013. IEEE Computer Society.

[27] S. Volckaert, B. Coppens, A. Voulimeneas, A. Homescu, P. Larsen, B. De Sutter, and M. Franz. Secure and efficient application monitoring and replication. In *USENIX Technical Conference*, 2016.

[28] S. Volckaert, B. De Sutter, T. De Baets, and K. De Bosschere. GHUMVEE: efficient, effective, and

flexible replication. In *Proc. Int'l Symp. on Foundations and practice of security*, 2013.

[29] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, CCS '12, pages 157–168, 2012.