Mixr: Flexible Runtime Rerandomization for Binaries

William Hawkins, Anh Nguyen-Tuong, Jason D. Hiser, Michele Co, Jack W. Davidson
Department of Computer Science, University of Virginia
{whh8b,an7s,jdh8d,mc2zk,jwd}@virginia.edu

ABSTRACT

Mixr is a novel moving target defense (MTD) system that improves on the traditional address space layout randomization (ASLR) security technique by giving security architects the tools to add "runtime ASLR" to existing software programs and libraries without access to their source code or debugging information and without requiring changes to the host's linker, loader or kernel. Runtime ASLR systems rerandomize the code of a program/library throughout execution at rerandomization points and with a particular granularity. The security professional deploying the Mixr system on a program/library has the flexibility to specify the frequency of runtime rerandomization and the granularity. For example, she/he can specify that the program rerandomizes itself on 60-byte boundaries every time the `write()` system call is invoked. The Mixr MTD of runtime ASLR protects binary programs and software libraries that are vulnerable to information leaks and attacks based on that information.

Mixr is an improvement on the state of the art in runtime ASLR systems. Mixr gives the security architect the flexibility to specify the rerandomization points and granularity and does not require access to the target program/library's source code, debugging information or other metadata. Nor does Mixr require changes to the host's linker, loader or kernel to execute the protected software. No existing runtime ASLR system offers those capabilities. The tradeoff is that applying the Mixr MTD of runtime ASLR protection requires successful disassembly of a program – something which is not always possible. Moreoever, the runtime overhead of a Mixr-protected program is non-trivial.

Mixr, besides being a tool for implementing the MTD of runtime ASLR, has the potential to further improve software security in other ways. For example, Mixr could be deployed to implement noise injection into software to thwart side-channel attacks using differential power analysis.

CCS CONCEPTS

• **Security and privacy → Embedded systems security**; **Software reverse engineering**; • **Software and its engineering → Compilers**; **Dynamic compilers**; **Software reverse engineering**;

ACM Reference Format:
William Hawkins, Anh Nguyen-Tuong, Jason D. Hiser, Michele Co, Jack W. Davidson. 2017. Mixr: Flexible Runtime Rerandomization for Binaries. In *Proceedings of MTD'17, Dallas, TX, USA, October 30, 2017,* 11 pages.
https://doi.org/10.1145/3140549.3140551

1 INTRODUCTION

Moving target defenses (MTDs) are an important area of security research [17, 22]. Among other uses, MTDs can protect software vulnerable to information leaks which enable attacks such as return-to-libc and return-oriented programming. In the context of an adversary that steals or infers information (in-memory passwords, cryptographic keys, locations of vulnerable functions, etc.) from a program and uses that information to build an attack, MTDs make the attacker's job more difficult by consistently changing the program in a way that invalidates the collected information and thereby prevents him/her from building a weaponized attack—a robust attack that has a high probability of succeeding.

This paper presents Mixr, a MTD for software of unknown provenance (SOUP). SOUP is software where the source code is not available and therefore the security of the software is uncertain. Mixr improves traditional address space layout randomization (ASLR). Whereas ASLR makes coarse-grained changes to the address space of a program once (at startup), runtime ASLR rerandomizes the address space of a program *throughout its execution* As a runtime-ASLR-protected program reaches rerandomization points throughout execution, its code is rearranged on pre-configured boundaries in a way that invalidates stolen information about the program that the adversary would otherwise be able to use to mount his/her attack. One of the particular benefits of Mixr is the flexibility it provides to the user to customize the security solution based upon the acceptable level of risk.

There are existing research projects that have attempted to extend ASLR in a similar fashion (See Table 3). None of the existing systems is able to protect SOUP adequately: some require recompilation of the vulnerable program; some require a modified version of the compiler; some require a modified linker, loader or shell; some require a modified kernel. Nor are any of the existing systems flexible enough to support the variety of security policies that balance the tradeoff between risk/security and performance/overhead: some mandate rerandomization points; some fix the boundaries at which the program is rerandomized; some only rerandomize at program startup.

Mixr is a system that can add the MTD of runtime ASLR to a software/library without access to its source code, debugging symbols or other metadata. A program secured with Mixr can run on the same platform as the vulnerable version without changes to the compiler, linker, loader or kernel. A Mixr user is able to choose the rerandomization intervals and boundaries upon which to rearrange program instructions. For example, a user can add runtime ASLR to a program that rerandomizes the vulnerable program at 60-byte boundaries on every execution of a `write()` system call.

Because Mixr is implemented using a static binary rewriter that relies on the successful disassembly of the potentially vulnerable target program/library, a task that is not decidable in the general case, there is the possibility that Mixr is not applicable to every artifact. Furthermore, the Mixr system is the non-trivial runtime overhead of a protected program.

Because of the flexibility of Mixr's design and implementation, it can also be used to implement other MTDs like artificial noise injection to prevent side-channel attacks based on differential timing analysis.

The remainder of this paper follows this outline: Section 2 explains why moving target, runtime ASLR defenses are an improvement on the state-of-the-art in software security; Section 3 formalizes the threats that runtime ASLR mitigates and Section 4 explains Mixr's design with respect to that threat model; Section 5 describes how Mixr works; Section 6 explores an alternate design and implementation of Mixr; Section 7 presents an evaluation of the performance of Mixr'd programs; Section 8 describes related research and the differences between existing runtime ASLR implementations and Mixr; Section 9 describes ways to continue work on Mixr and, finally, Section 10 concludes.

2 MOTIVATION

Years ago, the most devastating software attacks were so-called shellcode injection attacks. Advances in software security since those attacks were pioneered have largely mitigated their effects. Non-executable stacks, in particular, rendered most, if not all, of the shellcode injection attacks impotent.[1] In place of the code injection attacks, return-to-libc and ROP attacks prospered.

The return-to-libc and ROP attacks share the assumption that an attacker knows the location of parts of the vulnerable program in memory at runtime. In return-to-libc attacks, it is assumed that the attacker can learn the address of the target libc function. In ROP attacks, it is assumed that the attacker can learn the addresses of the ROP gadgets necessary to build the attack program. The ASLR techniques introduced by the Pax Security Team and researchers at the University of Illinois Urbana-Champaign provided an effective mechanism for invalidating this assumption and its implementation has rendered many return-to-libc and ROP attacks invalid [16, 25, 29].

Beyond advanced attacks like BROP [2] (and its successors JROP, etc.) that can defeat ASLR, there are fundamental reasons why ASLR is not always effective. First, many ASLR implementations only randomize the addresses of programs that the developer compiled with position independent code. Second, many implementations are limited by the constraints of the operating system's virtual memory system which means that ASLR cannot take advantage of all the entropy the system can offer for generating random addresses to place the code in memory at runtime. Third, common implementations only randomize the starting location for programs' sections. In other words, no matter how ASLR randomizes the location of the program's text section, the instruction locations within that section are always the same relative to one another. Fourth, ASLR implementations are not effective against programs that do just in time (JIT) compilation or can otherwise modify executable code at runtime (*e.g.*, [3]).

Despite these shortcomings, ASLR is very *nearly* a completely effective defense against modern attack techniques. "Runtime ASLR", a type of moving target ASLR that changes program layout throughout execution with fine granularity and high entropy, addresses the shortcomings identified with traditional ASLR and provides additional software protection.

3 THREAT MODEL AND DEFENSES

Mixr is designed specifically to limit an attacker's ability to maliciously use the information he/she can learn about the runtime status of a vulnerable program.

It is assumed that the adversary can learn "information" about the vulnerable program, including the contents of memory and/or the location of instructions in memory. This information is available to the attacker from an *oracle* (O) in the vulnerable program that gives responses to an attacker's *query* (q). The security architect attempting to secure the target program is aware of the parts of the vulnerable application that may be oracles.

In a typical attack scenario, oracles will exist at places in the vulnerable program where the attacker can trick the program into executing code that leaks information of their choosing. For example, for a vulnerable program on an x86 platform, an oracle may exist where the attacker can control `%rdi`, `%rsi`, and `%rdx` and then execute a `write()` system call. However, a program's vulnerability to malicious I/O is not the only way that an attacker can learn information about the program. An adversary could learn information by observing the timing of the program or the physical status of the processor as the program runs (*i.e.*, the adversary can employ side-channel analysis).

It is also assumed that the program contains a vulnerability through which an adversary has the power to use query responses to hijack program control. While it is assumed that the vulnerability allows the attacker to control program execution and write data anywhere in memory, it is assumed that the operating system upon which these vulnerable programs execute has protections against executing code injected onto the stack and into the heap.

The oracle's responses to the attacker's queries give him/her information required for attacking the vulnerable program. Learning enough information to mount a successful attack will require one or more queries. Equation 1 formalizes these notions.

$$\begin{aligned} response &= O(q) \\ t_{response} &= TimeOf(response) \\ t_{invalidate} &= TimeOf(Invalidate(response)) \\ t_{attack} &= TimeOf(Attack(response)) \end{aligned} \tag{1}$$

The period of time between when an attacker receives the oracle's response to a particular query and the time when that response is invalidated is known as the *attack window* (W). The attack window opens as soon as the adversary receives the response(s) required to mount an attack. The attack window closes as soon as those responses are invalidated (Equation 2).

[1]There are similar attacks where the attacker places malicious code into the heap and transfers program execution there. Although non-executable stacks cannot prevent this version of the attack, other mechanisms have been developed and deployed to defeat this variant.

$$W = [t_{response}, t_{response} + t_{invalidate}] \quad (2)$$

If the adversary cannot deploy an attack before the attack window closes at time $t_{response} + t_{invalidate}$, he/she cannot base their attack on that information. Invalidating oracle responses as quickly as possible through frequent appropriate runtime rerandomizations will decrease W and increase software security.

Invalidate is implemented with an *appropriate* runtime rerandomization that deliberately changes the program so that the adversary's assumptions with respect to the oracle's responses and their utility in mounting an attack are invalidated. In this context, an appropriate rerandomization is one done with enough granularity so that:

(1) responses that contained the address a of instructions i_1, i_2, . . ., i_n required to launch an attack no longer points to those instructions;
(2) responses that contained the address a of bytes $b_1, b_2, \ldots, b_n$ required to launch an attack no longer points to that data.

4 DESIGN

Mixr is designed to address the challenges of unpredictability, coverage and timeliness faced by any MTD [16]. Mixr is a system that allows the security architect to add appropriate invalidation points to potentially vulnerable software in order to minimize W and, consequently, improve its security. At a high level, to meet the demands and definitions in Section 3 and operate in the context outlined in Section 1, the Mixr system must meet the following requirements.

First, the system must be able to protect SOUP. Security architects are in a position where they must defend all applications no matter whether the program's source code is available or the software has debugging or metadata information embedded.

Second, the protected program must be able to run on the same platform as the original without modifications to the system's compiler, linker, loader or kernel.

Third, Mixr's design must not restrict the security architect's choice of which security policy to implement. As a consequence, the system must be optimized in a way that makes invalidations as lightweight as possible otherwise the security architect may not be able to implement his/her desired policy without excessive performance penalties. In Section 6, an alternate design is considered that demonstrates the tradeoffs of the design decision to optimize for lightweight rerandomizations.

Fourth, the system must be able to protect a program with granularity finer than basic blocks, functions, segments, or modules. The granularity is related to whether an invalidation is appropriate or not. As mentioned previously, one of the shortcomings of ASLR is that it randomizes the program segments as a unit. This means that an attacker can leverage information about the *relative* location of two instructions to craft his/her attack. As the granularity of the rerandomization increases, the less likely it is that the attacker can gain usable information about the relative location of instructions that may be useful in an attack. The granularity will affect runtime performance and must be an option for the user. A user more interested in security will specify a finer granularity while a user with more tolerance for risk may specify a coarser granularity in order to minimize performance overhead.

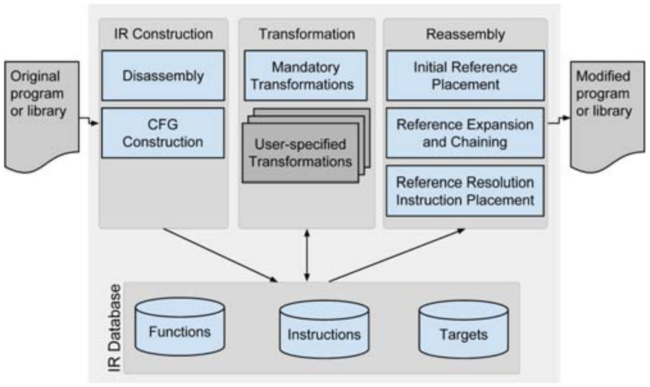
Original program or library
IR Construction
Disassembly
CFG Construction
Transformation
Mandatory Transformations
User-specified Transformations
Reassembly
Initial Reference Placement
Reference Expansion and Chaining
Reference Resolution Instruction Placement
Modified program or library
IR Database
Functions
Instructions
Targets

Figure 1: Overview of the operation of the Zipr static rewriter. Mixr is a plugin to Zipr that operates in the Transformation and Reassembly phases.

Mixr works in three phases. In the first phase, the preparation phase, the original program is rewritten to support runtime rerandomization. In the second phase, the linking phase, the original program is augmented with the components that implement randomization at runtime. Third, the rewritten program rerandomizes itself at runtime at the user-specified invalidation points.

The preparation phase relies on the user to specify the invalidation point(s) and the granularity. The native program is first organized into bundles whose size matches the user's choice of granularity. Each bundle is self-contained insofar as it contains all the information necessary to function correctly no matter where it exists in memory. Then the input binary is annotated to mark the user-chosen invalidation points. See Section 5 for the details of the bundles and the metadata contained in each bundle required to make them self-contained.

The linking phase also takes user input and adds the code necessary to implement the runtime randomization at invalidation points as the program executes. The user chooses one of the several different runtime randomization algorithms that perform the invalidations. See 5.2 for the details of these algorithms.

No matter what algorithm the user chooses for invalidation, it is the code added in the linking phase that is executed at runtime to perform runtime rerandomization. Because bundles are self contained, the implementation of the invalidation algorithms is straightforward. See Section 5.2 for details. Note that the code that implements runtime rerandomization is itself not rerandomized at runtime.

5 IMPLEMENTATION

Mixr is implemented as a plug-in for the Zipr static binary rewriter [15]. Zipr statically rewrites programs/libraries without access to their source code, debugging symbols or other metadata and offers an API and SDK for third parties to write extensions and customizations for Zipr's fundamental rewriting techniques. Zipr prepares a binary in three phases: disassembly and analysis, transformation and reassembly.

In the disassembly and analysis phase, Zipr analyzes a program/library to find its indirect branch targets (IBTs, called *pinned*

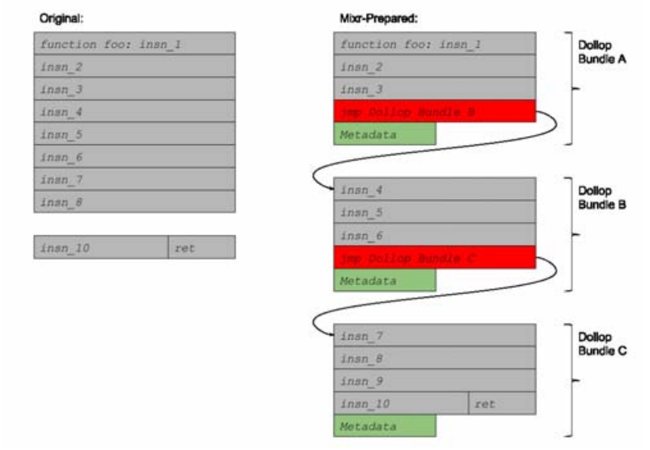

Figure 2: Creation of dollop bundles from straight line code in the original program.

addresses by Zipr) and recover the control flow graph. The intermediate representation (IR) recovered from disassembly and analysis is passed to the transformation stage where user-defined transformations change the program by altering its IR. Finally, the transformed IR is passed to the reassembly phase where a rewritten version of the program is generated. The reassembled program is executable/usable on the same platform as the original program without any additional runtime support.

Mixr is implemented using Zipr's APIs in the transformation and reassembly phase. Mixr uses the Zipr API to transform the program to include the necessary runtime support for rerandomization and the triggers for performing the rerandomization. Because Mixr is implemented using Zipr, Zipr's limitations apply to Mixr. Prior work [15] describes those limitations in detail.

Mixr uses the Zipr API during the reassembly phase to rewrite the program according to the specification described later in this section. Fundamental to Zipr's reassembly technique is the *dollop*. The dollop is like a basic block with multiple exit points. Dollops end at unconditional control flow transfer (a `ret`, `jmp`, etc). Every direct control flow transfer in a program statically rewritten by Zipr targets the head of another dollop.

Through its API, Zipr allows a third party to alter the definition of dollops and create them according to different criteria. Instead of a dollop being built strictly according to a program's control flow, Mixr builds fixed-size dollops known as *dollop bundles*. In Figure 2, the straight line sequence of code shown on the left is broken into three fixed-size dollop bundles, A, B and C. The dollop bundle contains enough metadata information so that it can be moved and rewritten according to that data without affecting its ability to execute. Figure 3 shows an idealized representation of the layout of a Mixr'd program at startup.

5.1 Metadata

5.1.1 Dollop Table. The dollop table is a contiguous area of memory that contains information about the current, randomized place of each dollop bundle in memory. For every dollop bundle, the table contains three pieces of information.

Address A pointer to the dollop bundle's place in memory.

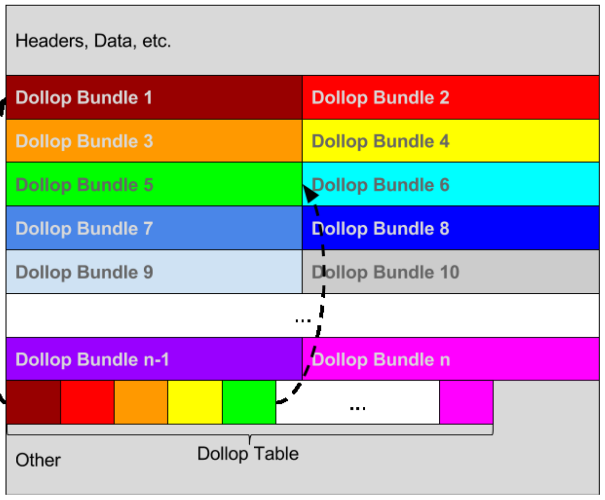

Figure 3: A Mixr'd program at startup.

RW Table Offset A pointer to the start of the dollop bundle's RW Table entry. This is a relative offset from the start of the dollop.

In-Use Flag A flag to indicate whether the dollop bundle is referenced from the program's runtime stack.

The value of all three fields is changed with every randomization. The table itself is not rerandomized at runtime but, because there is no identification between the dollop table's values and the target dollops, the adversary gains no leverage by leaking its values.

5.1.2 Dollop Bundles. For every dollop, there is a dollop bundle. The dollop bundle contains the dollop itself and the associated metadata needed to make the dollop self-contained. For any dollop that cannot be entirely contained within a dollop bundle, the dollop contains an added instruction (the *fallthrough* instruction) that links it to the subsequent dollop bundle. See Figure 2 where the instructions colored in red are fallthrough instructions. In order to completely contain a sequence of assembly instructions that can execute no matter where they are in memory at runtime, there needs to be additional information for any jump or fallthrough instruction and any instruction that makes a position-dependent memory reference. The dollop bundle contains a *read/write (RW) table* and a *dollop entry (DE) table.*

5.1.3 Read/Write Table. The RW table contains information needed to fixup the instructions in a dollop bundle that reference memory based on the location of the instruction in the program's address space, called position-dependent memory accesses.

Absolute Address The absolute address of the memory access by the position-dependent memory access instruction. The absolute address is calculated offline when the original binary is being prepared.

Instruction End The location of the end of the instruction to which this RW table entry applies.

Address Offset The location of the offset within the instruction itself. This is an offset from the end of the instruction.

Address Size The size of the address field in the position-dependent memory access instruction.

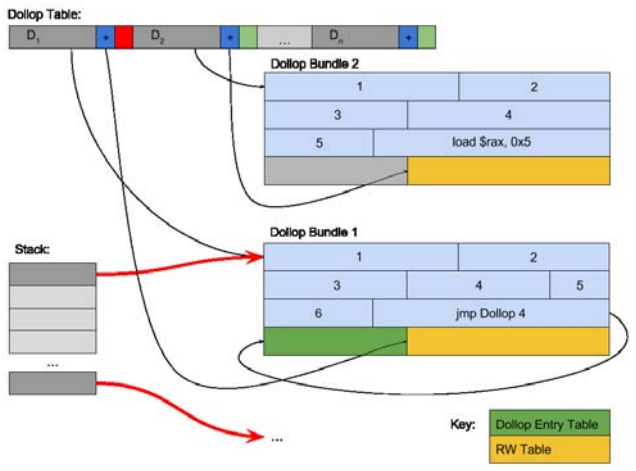

Figure 4: The links between the metadata components of a Mixr'd program.

5.1.4 Dollop Entry Table. The dollop entry (DE) table contains information that augments the jump instructions in a dollop bundle. Every jump in a program rewritten by Zipr is to the start of a dollop. Each entry in the DE table is itself a jump instruction that transfers program control indirectly to the target dollop through the address of the dollop bundle in memory as stored in the dollop table.

Figure 4 shows the links between the metadata components of a Mixr'd program described in the preceding subsections. Dollop Bundle 1 is Active. The first entry in the stack points to Dollop Bundle 1 and the in-use flag is set for its entry in the dollop table. Dollop Bundle 2, on the other hand, is not active. There are no links to it from the stack and, therefore, the in-use flag is not set in its entry in the dollop table. In Dollop Bundle 1 there is a `jmp` that points to an entry in Dollop Bundle 1's DE table. In Dollop Bundle 2, however, there are no `jmp` instructions and, therefore, the DE Table is empty. In reality, this would result in a dollop bundle that does not contain any space for the DE table. Figure 4 shows a space for educational purposes only. Dollop Bundle 2 has a position-dependent memory access instruction and, therefore, the RW table contains an entry. Dollop Bundle 1, on the other hand, does not have any position-dependent memory access instructions, and, therefore, the RW table is empty. Again, in reality a lack of position-dependent memory access instructions in the dollop bundle means that the dollop bundle does not contain any space for a RW table. The space is shown here for educational purposes only.

5.2 Rerandomization

The rerandomization algorithm is triggered every time the program reaches a user-defined invalidation point.[2] Before rerandomization occurs, the algorithm searches for the dollops referenced by the program's runtime stack and updates the in-use flag accordingly. Once this search is complete, rerandomization itself can happen.

The user has a choice of one of three rerandomization algorithms. Each rerandomization algorithm works fundamentally by

[2]Recall that the implementation of the rerandomization algorithm is not itself rerandomized at runtime.

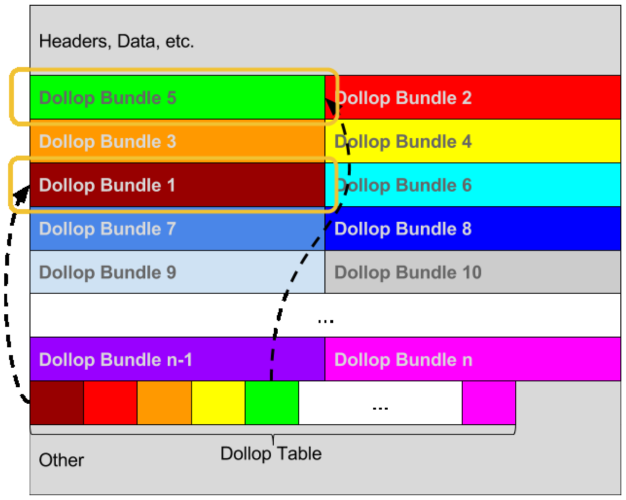

Figure 5: A Mixr'd program after a single rerandomization.

choosing two different dollop bundles to swap. The variety in these algorithms is with respect to how those pairs are selected. In the Random algorithm, pairs are chosen randomly and the user controls how many pairs of dollops are swapped at each rerandomization point. The default is 10. The Sequential algorithm walks the length of the dollop table and swaps each dollop bundle with its neighbor. In the Sequential Random algorithm every bundle in the program *a*, is swapped with a randomly chosen bundle *b*.

Once a pair of dollop bundles is chosen, a SWAP function swaps them. The definition is not shown in detail because of space constraints. At a high level, SWAP performs two operations. First, it exchanges the two dollop bundles in memory. Second, it walks the RW table and rewrites the position-dependent memory operations as necessary. Each position-dependent memory operation is rewritten so that it addresses the correct absolute address based on its updated location within the program after rerandomization.

Figure 5 shows an idealized representation of a Mixr'd program after a single shuffle. In this figure, Dollop Bundles 1 and 5 have been swapped. Note that the entries for Dollop Bundles 1 and 5 in the dollop table are updated to reflect the new location of those dollop bundles in memory.

6 ALTERNATE IMPLEMENTATION

As mentioned in Section 4, one of the design goals of Mixr was to make rerandomization efficient and simple to implement. This goal led to the decision to make every dollop bundle self contained. For all but instructions that make position-dependent memory references, the resulting design made it possible to accomplish rerandomization with simple memory copies.

Instructions that make position-dependent memory references are not the only instructions that require special handling if they are to be completely encapsulated in a bundle. In the original design, jump instructions are modified so that their target is always an entry in the dollop bundle's metadata. Those targets then implement the position-independent transfer of program control that takes into account the current position of the target dollop in the program's address space (See Section 5.1.4).

This method for encapsulating jump instructions in the dollop bundle trades efficiency of rerandomization for overhead at runtime when those instructions are executed. This tradeoff makes it possible for the Mixr user to specify frequently executed invalidations without worrying about performance impact. This tradeoff was explicitly recognized in the original design but is not a choice that every user of Mixr may want.

Therefore, the Mixr user may benefit from an alternate technique for dollop bundle encapsulation that trades overhead at rerandomization for efficiency at runtime. In use cases where Mixr is deployed on software where rerandomizations occur infrequently, the benefit of using this alternate technique may be substantial.

An alternate method treats jump instructions in the same way that the original encapsulation method treats instructions that make position-dependent memory references. There is still a DE table but the entries contain different information. Instead of each entry being itself a jump instruction that redirects program control to the target dollop through an entry in the dollop table, the entry contains only the location of the target dollop's entry in the dollop table. That entry is used during rerandomization to rewrite the jump instructions in the dollop bundle according to the new location of the dollop in memory.

In practice, this requires very little change to the specifics of the bundle encapsulation but does result in space savings. The entry in the DE table in the alternate version contains two entries.

Instruction End The dollop-relative location of the end of the instruction to which this DE table entry applies.

Dollop Table Pointer The address of this DE table entry target dollop's entry in the dollop table.

In addition to changes to the dollop bundle's encapsulation, the alternate design and implementation has implications for the runtime rerandomization process, too. Jump instructions are now treated like instructions that make position-dependent memory references.

The ALT_SWAP algorithm is not shown in detail because of space constraints. At a high level, ALT_SWAP performs three operations and the first two are identical to the steps of the SWAP function. An additional step in the ALT_SWAP algorithm walks the DE table and fixes the jump instructions as necessary. The target of each jump instruction is rewritten according to the current location of the target in memory as stored in the dollop table.

7 RESULTS

7.1 Security

Query responses from oracles give adversaries information they need to mount attacks (Section 2). In runtime rerandomization systems, responses are invalidated when the program is rerandomized at runtime in an appropriate manner. Runtime rerandomization systems that do not change the underlying program with fine enough granularity are susceptible to attacks using leaked information about relative location of addresses within rerandomization units. For this reason, the granularity of rerandomization is important.

Table 1 shows the average sizes, in bytes, of the functions and basic blocks of the programs in the SPEC benchmark suite with various compiler optimization settings. The results were gathered with version 4.8.4 of the gcc compiler. STABILIZER [9], Marlin [14] and selfrando [6] rerandomize at function boundaries while STIR [26], Remix [5] and Chronomorph [12] rerandomize at basic block boundaries.[3] No matter whether the system rerandomizes at basic block or function boundaries, the compiler determines the granularity. More importantly, the sizes of functions under certain configurations are large enough to contain a significant number of gadgets that an attacker may use to launch an attack. For instance, the version of `bzip2` compiled for the SPEC2006 benchmark has 25 functions with at least 10 ROP gadgets. On average, every function in `gcc` contains more than 7 gadgets and one function contains more than 90.

	Functions		Basic Blocks	
	Count	Average Size	Count	Average Size
O0	91342	477.40052	974715	44.737917
O2	48991	609.765937	832713	35.874356

Table 1: Average basic block and function sizes, in bytes, for the programs in the SPEC benchmark suite when compiled with different optimization levels using gcc version 4.8.4.

	Bundles	
Granularity	Count	Average Size
60	1367454	25.9827
120	900866	36.848951
240	792756	41.191778

Table 2: Average code sizes per bundle in the Mixr'd versions of the programs in the SPEC benchmark suite with different granularities.

Because of the required metadata to make dollop bundles self contained, in a Mixr'd program with, for example, 60 byte granularity, each bundle will not necessarily contain 60 bytes of instructions. Table 2 shows the amount of instructions, in bytes, contained in each dollop bundle on average for the programs in the SPEC benchmark suite. In the Mixr'd version of `bzip2` for the SPEC benchmark suite (with a 60 byte granularity), no dollop bundle contains more than 9 gadgets. On average, each dollop bundle in `bzip2` contains less than a single gadget.

These results show that a Mixr'd program is able to perform runtime rerandomization with a much finer granularity than existing runtime rerandomization systems.

7.2 Performance

There are three sources of overhead that result from Mixr and these overhead sources correspond roughly to the three phases of Mixr:

[Source 1] Preparation The self-contained dollop bundles introduce an additional jump operation for every jump operation in the original program. See 5.1 for the details. More importantly, breaking the optimized straight line code produced by the compiler into a series of dollop bundles introduces fallthrough instructions to link the dollop bundles back together. See Figure 2. These fallthrough instructions add significant performance overhead at runtime.

[3]The other runtime randomization systems rerandomize with much less granularity, either page or segment boundaries.

[Source 2] Linking Linking the rerandomization implementation to the Mixr'd program increases the size of the program on disk and may affect the memory usage of the Mixr'd program at runtime.
[Source 3] Rerandomization Rerandomization adds overhead at runtime every time an invalidation point is reached.

The SPEC2006 benchmark suite was used to evaluate the performance of a Mixr'd program.[4] The host used to collect data for these experiments has a single CPU and 48GB of RAM. The CPU, an Intel Xeon CPU E5645, runs at 2.40GHz and has 6 cores. Each core has two threads for a hostwide total of 12 threads. The CPU has 12MB of cache. The compiler used to generate the native versions of the Mixr'd programs is GCC version 4.8.4. The operating system is the 14.04 LTS distribution of Ubuntu Linux.

The left, darker bar in Figure 6 shows the results of an evaluation of the first source of overhead in a Mixr'd program (with granularity set to 60 bytes). To isolate the performance overhead of the runtime rerandomization (Source 3) from the overhead of the preparation process itself (Source 1), only one invalidation point was chosen for each benchmark – the `start` function. Rerandomization was performed according to the Sequential algorithm.

On average, a Mixr'd program exhibits 2.25x performance overhead at runtime. By virtue of the experiment's organization, it is possible to assign this overhead to two sources. The first source is the static rewriting process itself. See [15] for the overhead attributable to the rewriting process itself. The remaining overhead is from Source 1.

The variation in performance overhead among the programs in the benchmark suite is due to the division of the program's code into bundles. First, for programs in the benchmark suite that contain a significant number of jump operations in their computationally intensive kernels, the overhead is significantly higher because of the way that the bundles are designed to be self-contained. Second, for programs in the benchmark that have long, straight line sequences of code, the division of those sequences into a series of dollop bundles introduces the overhead from the repeated execution of fallthrough instructions.

There are two measures of Source 2 overhead – on-disk file size of a Mixr'd program and the maximum RSS of a Mixr'd program at runtime. The difference in size between an original, input program and the Mixr'd version of the program is caused by the space required for metadata (Section 5.1), the code added to the Mixr'd program to implement the rerandomization algorithms (Section 5.2) and the overhead intrinsic to the static rewriting process ([15]).

The size of the code required for the implementation of the runtime rerandomization algorithms is constant. The Sequential runtime rerandomization algorithm, for example, is implemented in 16 kilobytes. No effort has been made to optimize this implementation for size.

The size of the metadata required by Mixr preparation is determined by a) the number of bundles in the program, b) the number of jumps in the program and c) the number of position-dependent memory operations. See Section 5.1 for details.

On average, maximum RSS overhead is 5.4% and file size overhead is 3.51x.

[4]One benchmark, dealII did not compile on the test system. With the default configuration, gamess did not verify on the test system. Because these failures are independent of Mixr, the results are omitted.

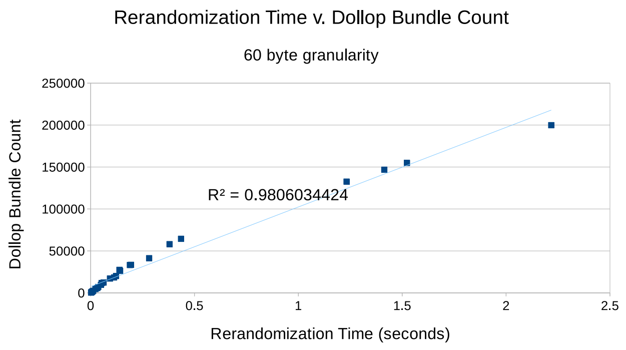

Figure 7: Rerandomization time plotted against the number of dollops in the Mixr'd program when prepared using the original design and implementation.

The dollop table for the, on average, approximately 38421 bundles for each program in the SPEC benchmark suite when the programs are Mixr'd with 60 byte granularity requires, on average, approximately 422 kilobytes. On average, there are 48278 instructions that require entries in the DE tables. The DE tables, on average, require 337 kilobytes of space for each of the programs. On average, there are approximately 4769 position-dependent memory operations in each program in the SPEC benchmark suite when the programs are Mixr'd with 60 byte granularity. The RW tables, on average, require approximately 987 kilobytes for each of the programs. In total, the metadata for each SPEC program amounts to, on average, approximately 1.74 megabytes. For reference, the average size of each of the programs in the SPEC benchmark suite is approximately 1.27 megabytes.

It is difficult, if not impossible, to find a set of invalidation points in the programs in the SPEC2006 benchmark suite that would allow for a consistent, systematic analysis of the overhead from Source 3. Therefore, an alternate experiment is required. Each program in the SPEC benchmark suite was Mixr'd with the Sequential rerandomization algorithm and the special configuration to rerandomize 500 times at program startup. Once the benchmark application has performed its self-randomization 500 times, the program terminates. A comparison execution was performed where the programs rerandomized 1000 times. The difference in the performance of each of the benchmarks for 500 rerandomization at startup and for 1000 rerandomizations can be used to deduce the overhead of randomizations. The results are shown in Figure 7.

There is a very strong linear correlation ($r^2 = 0.98$) between the number of dollop bundles in a program and the time it takes to perform a single rerandomization. This is expected considering that SWAP is O(n) where n is the number of dollops in the Mixr'd program.

7.2.1 *Performance of Alternate Design and Implementation.*

As explored in Section 4, there is a certain amount of extra overhead in a design focused on optimizing for rerandomizations. Section 6 described an alternate to the original design that is optimized to minimize the overhead of a Mixr'd program in steady state.

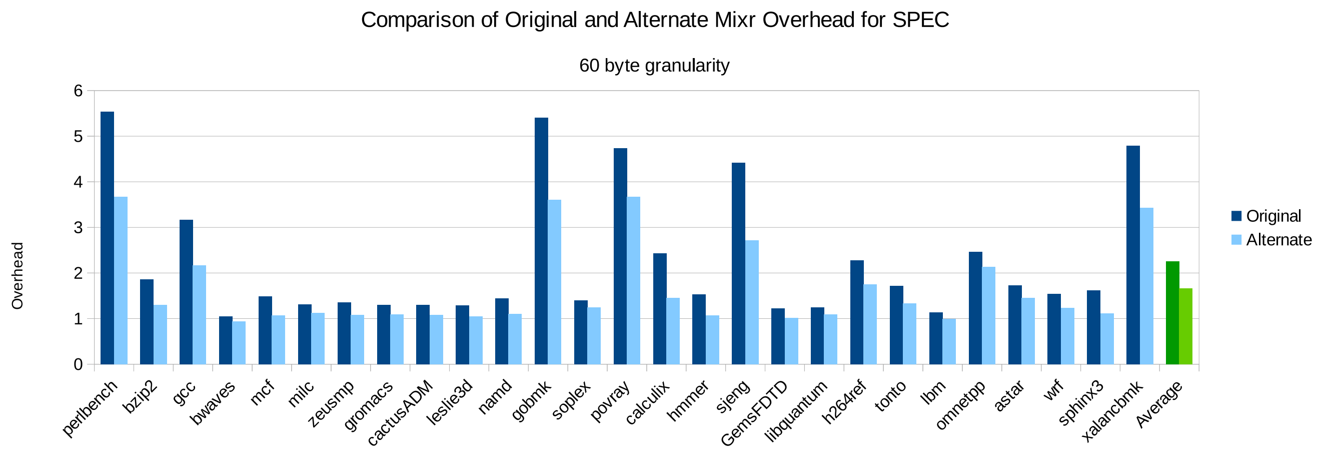

Figure 6: Comparing the Mixr runtime overhead for the programs of the SPEC2006 benchmark suite when prepared using the original and alternate design and implementation.

The following experiments evaluate the performance of this alternate design. The right hand, lighter bars in Figure 6 shows the results of this experiment.

On average, a Mixr'd program that follows the alternate design exhibits 1.66x performance overhead at runtime. That is a 30.17% improvement over the original design. In other words, the alternate design and implementation does exhibit the hypothesized performance improvement over the original design and implementation.

On average, maximum RSS overhead for the alternate design is 5.5% with a file size overhead of 3.58x. Those overheads are roughly equivalent to the maximum RSS and file size overheads of the original design.

7.2.2 Performance Overhead and Granularity. In both the original and the alternate preparation techniques, it is expected that changes to the granularity will affect performance. Figure 8 shows the results of an experiment designed to test whether varying granularity affects the performance overhead for Mixr'd programs using the alternate technique.

There are two reasons why the performance improves as the granularity decreases. First, as the granularity decreases, there are fewer dollop bundles. Recall that in this experimental configuration at the beginning of each of the benchmarks, the Mixr'd program rerandomizes a single time using the Sequential rerandomization algorithm. Therefore, as the number of dollop bundles decreases there are fewer operations to complete at the start of every benchmark to rerandomize and implies that the performance of the benchmark will improve, albeit slightly.

Second, and of more consequence to performance overhead, as the granularity decreases, the Mixr'd program is able to execute more straight line code before having to execute a fallthrough instruction to reach the subsequent dollop bundle. Although the results seem to indicate that there is a lower bound for the performance improvement gained by decreasing granularity, this is not the case. Note that as the granularity decreased from 120 to 240, each dollop bundle contained only 10% more instructions (See Table 2). The proportional difference between the additional size of the dollop bundle and the amount of code contained in each is the result of the additional metadata that needs to be captured to make each dollop bundle self-contained. Future test results will help determine whether this is a valid conclusion and whether additional performance improvements can be gained by decreasing granularity. See Section 9.

8 RELATED WORK

There have been numerous, promising research projects on MTDs. Some deploy offline diversification (*e.g.*, [24]), online control flow randomization (*e.g.*, [7, 10]), offline or program-startup coarse- and fine-grained ASLR (*e.g.*, [8, 11, 19, 21, 29]) or randomization of an entire ISA (*e.g.*, [18]). Table 3 summarizes only the research projects immediately comparable to Mixr. Due to space limitations, we restrict the extended discussion below to a subset of the systems in Table 3.

RuntimeASLR is a runtime ASLR system that rerandomizes servers at the point that they `fork()` worker processes to handle client connections [23]. By rerandomizing at this point, the designers of RuntimeASLR minimize the performance overhead for those processes that actually perform work. Rerandomization in RuntimeASLR occurs at module boundaries âĂŞ similar to traditional implementations of ASLR. That randomizations occur at every `fork()` rather than a single time at program startup adds additional entropy with which the attacker must contend to launch their attack improves upon traditional ASLR but does not mitigate against information leaks from long-running worker processes. In RuntimeASLR the controlling server process that monitors and launches worker threads runs under Pin. After rerandomizing, Pin disconnects and the child processes run at native speeds. Servers implemented according to this model exhibit limited overhead under RuntimeASLR but servers that do follow this paradigm exhibit performance overhead from 217x to more than 110000x for certain SPEC applications.

CodeArmor is a runtime ASLR system that rerandomizes the program at segment boundaries whenever the program executes a

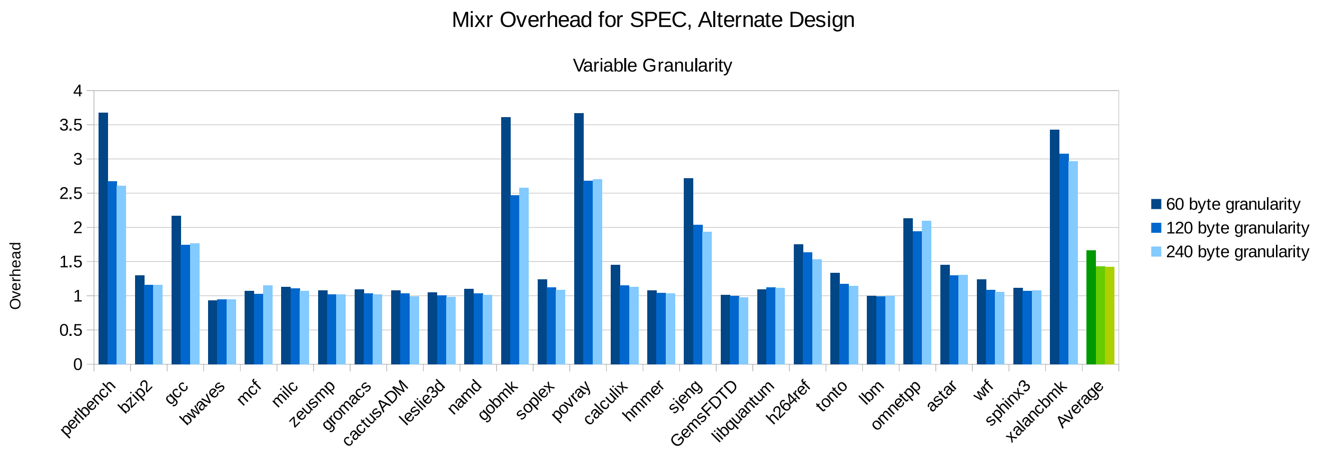

Figure 8: Changes in the performance overhead of a Mixr'd program as the granularity decreases.

Tool	Access to Source Code	Compiler Modification	Kernel Modification	Linker Modification	Loader Modification	Granularity	Invalidation Points
Mixr						User-specified	User-specified
Islands [28]	✗				✗	Function	User-specified
Remix [5]	✗	✗				Basic Block	User-specified
RuntimeASLR [23]						Modules	`fork()`
TASR [1]	✗	✗	✗	✗	✗	Segment	I/O system calls
CodeArmor [4]					✗	Segment	system calls
STABILIZER [9]	✗	✗				Functions	Every 500ms
Chronomorph [12]	✗					Basic Blocks	User-specified
Shuffler [27]	✗[1]				✗	Functions	Every 50ms
OS ASR [13]	✗	N/A				Function	User-specified

[1] Source code not required if the target binary program is compiled to save debugging information, symbol tables and relocations. Otherwise, recompilation is required.

Table 3: Comparison of requirements and features of each of the available runtime rerandomization technologies.

system call [4]. CodeArmor also rerandomizes the data segments of a program and inserts traps throughout the program to prevent the attacker from exploiting "lucky guesses" to hijack program control. CodeArmor does work on binary programs but requires the use of a customized loader. Based on the SPEC2006 benchmark and a set of tests against nine common server applications, CodeArmor'd programs operate 3.2% and 8.2% slower, respectively, and have an increased RSS of 4.4% and 13.4%, respectively.

Shuffler is a runtime ASLR technique that rerandomizes the program during execution at function boundaries once every 50ms [27]. Shuffler is designed to prevent code reuse attacks, especially those that cripple vulnerable programs with ROP, BROP and JIT-ROP attacks. At the same time that Shuffler protects a target program, the Shuffler runtime support system protects itself. Shuffler protects programs without requiring changes to the host's compiler or kernel. It does, however, require that the target application have preserved debugging and metadata information (DWARF debugging information, symbol tables and relocations) and that the host use a modified loader. According to tests run on the SPEC2006 benchmark suite, Shuffler adds 14.9% overhead.

9 FUTURE WORK

There are interesting extensions to Mixr that would make it more widely applicable. Most important is an extension to support rerandomizing the rerandomization algorithm itself which would add self-protection the way that Shuffler does [27]. As important is an extension to support multithreaded programs. Adding multithreading support would require that the Mixr rerandomization process walk each of the thread's stacks, and carefully avoid race conditions with executing threads. In addition, if the program that is being randomized copies a return address into a register or other memory, then Mixr may need to treat that like an address on the stack and skip the enclosing dollop bundle during re-randomization. This might increase re-randomization overhead.

Optimizations to the Mixr implementation to improve performance are also particularly important. Prior to implementing performance improvements, however, it is important to verify the hypothesis that decreasing granularity improves performance. Whether or not this is the case will determine which possible improvements to implement and which possible improvements to skip.

There are myriad extensions to Mixr like selective rerandomization and noise injection that would increase its value in the field of software security beyond its ability to add moving target runtime ASLR. In selective randomization, the security architect would choose only the most sensitive parts of the application (*e.g.*, the cryptographic code) to rerandomize at runtime which would improve the security for highly sensitive areas of the program and improve performance of the program when it is executing less critical sections of code. In noise injection, the rerandomization process could increase the variability of the CPU profile in a way that prevents side-channel attacks using differential power analysis [20].

10 CONCLUSION

Mixr is a system that can add the MTD of runtime ASLR in a way that existing runtime ASLR systems cannot. Mixr works on software/libraries without access to their source code, debugging symbols or other metadata. A program secured with Mixr can run on the same platform as the vulnerable version without changes to the compiler, linker, loader or kernel. Because the Mixr user is able to customize the rerandomization intervals and the boundaries upon which to rearrange program instructions, it provides a flexibility that the other systems do not. This flexibility represents a tradeoff. The possibility exists that the potentially vulnerable target application cannot be accurately disassembled making Mixr's protections inapplicable. Furthermore, the runtime overhead of a Mixr'd program is non-trivial.

Although the focus of this paper is on Mixr's ability to add a moving target runtime ASLR defense to a vulnerable program/library, Mixr is not limited to this application. In the future, extensions to Mixr can be used to implement selective runtime randomization and artificial noise injection.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous referees for their valuable comments and helpful suggestions. The work is supported by the Air Force Research Laboratory and Defense Advanced Research Projects Agency under Grant Nos.: FA8750-15-C-0118 and FA-8750-14-C-0110 and the Air Force Research Laboratory under Grant No.: FA8750-17-2-0079.

REFERENCES

[1] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. 2015. Timely Rerandomization for Mitigating Memory Disclosures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM Press, New York, New York, USA, 268–279.

[2] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazieres, and Dan Boneh. 2014. Hacking Blind. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP '14)*. IEEE Computer Society, Washington, DC, USA, 227–242.

[3] Dionysus Blazakis. 2010. Interpreter Exploitation. In *Proceedings of the 4th USENIX Conference on Offensive Technologies (WOOT'10)*. USENIX Association, Berkeley, CA, USA, 1–9.

[4] Xi Chen, Herbert Bos, and Cristiano Giuffrida. 2017. CodeArmor: Virtualizing the Code Space to Counter Disclosure Attacks. In *2017 IEEE European Symposium on Security and Privacy (Euro S&P)*. IEEE, Paris, France, 514–529.

[5] Yue Chen, Zhi Wang, David Whalley, and Long Lu. 2016. Remix: On-demand Live Randomization. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy (CODASPY '16)*. ACM, New York, NY, USA, 50–61.

[6] Mauro Conti, Stephen Crane, Tommaso Frassetto, Andrei Homescu, Georg Koppen, Per Larsen, Christopher Liebchen, Mike Perry, and Ahmad-Reza Sadeghi. 2016. Selfrando: Securing the Tor Browser against De-anonymization Exploits. *Proceedings on Privacy Enhancing Technologies* 2016, 4 (2016), 454–469.

[7] Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Fraz. 2015. Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity. In *2015 Network and Distributed System Security*. Internet Society, San Diego, CA. https://www.sba-research.org/wp-content/uploads/publications/ndss15b.pdf

[8] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. 2015. Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 763–780. https://doi.org/10.1109/SP.2015.52

[9] Charlie Curtsinger and Emery D. Berger. 2013. STABILIZER: Statistically Sound Performance Evaluation. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM Press, New York, New York, USA, 219–228.

[10] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z. Snow, and Fabian Monrose. 2015. Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming. In *The 2015 Network and Distributed System Securit*. Internet Society, San Diego, CA. https://www.internetsociety.org/sites/default/files/05

[11] Lucas Vincenzo Davi, Alexandra Dmitrienko, Stefan Nürnberger, and Ahmad-Reza Sadeghi. 2013. Gadge me if you can. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security - ASIA CCS '13*. ACM Press, New York, New York, USA, 299. https://doi.org/10.1145/2484313.2484351

[12] Scott Friedman, David Musliner, and Peter Keller. 2015. Chronomorphic Programs: Runtime Diversity Prevents Exploits and Reconnaissance. *International Journal on Advances in Security* 8, 3-4 (2015), 120–192.

[13] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. 2012. Enhanced operating system security through efficient and fine-grained address space randomization. (2012), 475–490 pages.

[14] Aditi Gupta, Javid Habibi, Michael S. Kirkpatrick, and Elisa Bertino. 2015. Marlin: Mitigating Code Reuse Attacks Using Code Randomization. *IEEE Transactions on Dependable and Secure Computing* 12, 3 (May 2015), 326–337.

[15] William H. Hawkins, Michele Co, Jason D. Hiser, Anh Nguyen-Tuong, and Jack W. Davidson. 2017. Zipr: Efficient Static Binary Rewriting for Security. In *Proceedings of the 47th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2017)*. IEEE, Denver, CO.

[16] Thomas Hobson, Hamed Okhravi, David Bigelow, Robert Rudd, and William Streilein. 2014. On the Challenges of Effective Movement. In *Proceedings of the First ACM Workshop on Moving Target Defense - MTD '14*. ACM Press, New York, New York, USA, 41–50. https://doi.org/10.1145/2663474.2663480

[17] Sushil Jajodia, Anup K. Ghosh, Vipin Swarup, Cliff Wang, and X. Sean Wang (Eds.). 2011. *Moving Target Defense: Creating Asymmetric Uncertainty for Cyber Threats*. Advances in Information Security, Vol. 54. Springer New York, New York, NY.

[18] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. 2003. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communication security - CCS '03*. ACM Press, New York, New York, USA, 272. https://doi.org/10.1145/948109.948146

[19] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. 2006. Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*. IEEE, 339–348. https://doi.org/10.1109/ACSAC.2006.9

[20] Paul Kocher, Joshua Jaffe, and Benjamin Jun. 1999. Differential Power Analysis. In *Advances in Cryptology - CRYPTO '99* (1 ed.), Michael Wiener (Ed.). Springer-Verlag Berlin Heidelberg, 388–397.

[21] Byoungyoung Lee, Long Lu, Tielei Wang, Taesoo Kim, and Wenke Lee. 2014. From Zygote to Morula: Fortifying Weakened ASLR on Android. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 424–439. https://doi.org/10.1109/SP.2014.34

[22] Peng Liu and Cliff Wang. 2016. MTD 2016: Third ACM Workshop on Moving Target Defense. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 1868–1869. https://doi.org/10.1145/2976749.2990483

[23] Kangjie Lu, Stefan Nürnberger, Michael Backes, and Wenke Lee. 2016. How to Make ASLR Win the Clone Wars: Runtime Re-Randomization. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS 2016)*. The Internet Society, San Diego, CA.

[24] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. 2012. Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code

Randomization. In *2012 IEEE Symposium on Security and Privacy*. IEEE, 601–615. https://doi.org/10.1109/SP.2012.41

[25] Pax Team. 2003. PaX ASLR Design and Implementation. (2003). https://pax.grsecurity.net/docs/aslr.txt

[26] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. 2012. Binary stirring. In *Proceedings of the 2012 ACM conference on Computer and communications security - CCS '12*. ACM Press, New York, New York, USA, 157. https://doi.org/10.1145/2382196.2382216

[27] David Williams-King, Graham Gobieski, Kent Williams-King, James P. Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P. Kemerlis, Junfeng Yang, and William Aiello. 2016. Shuffler: Fast and Deployable Continuous Code Re-randomization. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 367–382.

[28] Haizhi Xu and Steve J. Chapin. 2006. Improving address space randomization with a dynamic offset randomization technique. In *Proceedings of the 2006 ACM symposium on Applied computing - SAC '06*. ACM Press, New York, New York, USA, 384. https://doi.org/10.1145/1141277.1141364

[29] Jun Xu, Z. Kalbarczyk, and R.K. Iyer. 2006. Transparent runtime randomization for security. In *Proceedings of the 22nd International Symposium on Reliable Distributed Systems (SRDS'06)*. IEEE Comput. Soc, 260–269. https://doi.org/10.1109/RELDIS.2003.1238076