Timely Rerandomization for Mitigating Memory Disclosures*

David Bigelow
MIT Lincoln Laboratory
dbigelow@ll.mit.edu

Thomas Hobson
MIT Lincoln Laboratory
thomas.hobson@ll.mit.edu

Robert Rudd
MIT Lincoln Laboratory
robert.rudd@ll.mit.edu

William Streilein
MIT Lincoln Laboratory
wws@ll.mit.edu

Hamed Okhravi
MIT Lincoln Laboratory
hamed.okhravi@ll.mit.edu

Abstract

Address Space Layout Randomization (ASLR) can increase the cost of exploiting memory corruption vulnerabilities. One major weakness of ASLR is that it assumes the secrecy of memory addresses and is thus ineffective in the face of memory disclosure vulnerabilities. Even fine-grained variants of ASLR are shown to be ineffective against memory disclosures. In this paper we present an approach that synchronizes randomization with potential runtime disclosure. By applying rerandomization to the memory layout of a process every time it generates an output, our approach renders disclosures stale by the time they can be used by attackers to hijack control flow. We have developed a fully functioning prototype for x86_64 C programs by extending the Linux kernel, GCC, and the libc dynamic linker. The prototype operates on C source code and recompiles programs with a set of augmented information required to track pointer locations and support runtime rerandomization. Using this augmented information we dynamically relocate code segments and update code pointer values during runtime. Our evaluation on the SPEC CPU2006 benchmark, along with other applications, show that our technique incurs a very low performance overhead (2.1% on average).

1. INTRODUCTION

Memory corruption attacks have been one of the most prevalent types of attack for decades [5], and they continue to pose a threat to modern systems. These attacks have evolved from simple stack-based buffer overflows [33] to a more sophisticated type that reuses existing code in a process's memory space [40]. Known as code reuse attacks, or return-oriented programming (ROP), these attacks bypass traditional defenses such as marking memory pages writable or executable but not both (W⊕X), or defenses that check the integrity of code before execution [34]. Although in many cases these attacks may be mitigated through the use of a memory safe language, protecting against such attacks remains challenging due to enormous volumes of existing code combined with modern programs that continue to be developed in C/C++ [44] for reasons of practicality, performance, and developer familiarity.

Because memory safe languages are not always a reasonable option, numerous defenses have been proposed over the years to mitigate memory corruption attacks in non-memory safe languages. These defenses can be broadly categorized into enforcement-based and randomization-based defenses. In the enforcement-based category, certain checks are performed on the code to prevent memory corruption attacks. Complete memory safety techniques such as SoftBound and its CETS extension [31, 32] are an example of such defenses; however, they incur a high overhead, up to 4x slowdowns in some cases. Other enforcement-based techniques include control-flow integrity (CFI) [3], along with its coarse-grained enforcement variants [47, 48], that try to stop control-hijacking attacks by verifying the target of any control transfer. Unfortunately, ideal CFI has not proven practical to date, and coarse-grained enforcement of CFI has been shown to be ineffective [18].

In the randomization-based category, the process's instructions [26, 8] or memory layout [35] are randomized to thwart memory corruption attacks. A widely deployed memory randomization technique is Address Space Layout Randomization (ASLR), implemented in most modern desktop and mobile operating systems, that randomizes the location of the stack, heap, linked libraries, and main program [35]. A major assumption of randomization-based defenses is that the memory layout remains secret. Unfortunately, this assumption has been shown to be incorrect. Memory data can be leaked to a potential attacker either directly using memory disclosure vulnerabilities [43], or indirectly using remote timing or fault analysis attacks [38, 9]. In response to such weaknesses, one trend in the research community has been to make randomization techniques finer-grained. Techniques such as Binary Stirring [46], medium- and fine-grained ASLR [27, 17], or in-place code rewriting [20] try to make randomization of memory layout more granular than traditional memory segments. However, as shown by recent offensive techniques such as just-in-time ROP [42] or side-channel attacks [38], even fine-grained randomization can be bypassed with an extensive-enough leakage of memory content.

*This work is sponsored by the Department of Defense under Air Force Contract #FA8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the author and are not necessarily endorsed by the United States Government.

CCS'15, October 12–16, 2015, Denver, Colorado, USA.
 ISBN 978-1-4503-3832-5/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2810103.2813691.

In this work, we take a different approach to mitigate memory leakage attacks. We observe that the weaknesses of existing traditional and fine-grained randomization techniques arise from the fact that randomization happens a single time at program load and never again, while leaks may happen many times at runtime. As a result, the information leaked to an attacker remains valid as long as the process is running. We therefore rerandomize a process's memory layout at runtime in order to mitigate the impact of information leakage.

An important question is when to rerandomize the memory layout. Overly frequent rerandomization can incur an unacceptably high overhead, while insufficiently frequent rerandomization can weaken or eliminate security guarantees because an attacker may have the opportunity to leak memory layout information and execute the main attack before the process is rerandomized. A key insight in our technique is the need to tie rerandomization to the actions of a potential attacker. Therefore, we rerandomize the process memory layout whenever there is an *output* from the process; that is, a socket write, a file write, a console write, etc. More precisely, we rerandomize the memory layout after many output system calls and before processing an input system call. This mitigates the impact of memory disclosure attacks because any information about the layout of memory that an attacker obtains at output time is stale by the time there is an opportunity to use that information to hijack control flow (i.e. at input time).

In this paper, we describe the design, implementation, and evaluation of our technique, Timely Address Space Randomization (TASR[1]), which rerandomizes the memory layout during runtime before the attacker can take advantage of any stolen knowledge (hence, the *timely* aspect). TASR is conceptually simple, but its realization required a substantial design and implementation effort, most notably in tracking the locations of all code pointers in a live process. TASR consists of three major components. First, the compile-time component annotates source code written in C with information needed to support its relocation at runtime. This information is added as a new section into the compiled binary (ELF) file of the application which is used in a TASR-enabled system and ignored otherwise. Second, the kernel component of TASR uses this information to manage the rerandomization whenever the appropriate input/output system call pairings occur, and third, an injected process element performs the actual pointer updating each time it is required.

TASR was designed with the intent of achieving source code compatibility [44] with complex, production-level application while having low performance overhead, which has been shown to be critical for widespread adoption of defensive techniques.

Pointer analysis similar to what is needed for TASR has been studied before in other contexts including control-flow integrity techniques [3] or garbage collection for C [36]; however, those analyses are not sufficient for our prototype because the pointer analysis must be exact and complete in order for TASR to work properly. Imprecise pointer tracking can be tolerated in previous work where correctness is not at stake (e.g. a garbage collector simply fails to collect an unused region of memory which does not break functionality), but can cause crashes or worse in a TASR-enabled process. Therefore, we have also improved pointer tracking for compiler-generated temporary variables so as to achieve greater precision. We additionally use techniques from related work [36] to handle other hard cases, such as disambiguation of unions and other dynamically allocated objects.

TASR is practical and lightweight. We have implemented a complete TASR prototype which includes the improved compile-time pointer analyzer and a runtime rerandomizer on x86_64 Linux systems. We also evaluate the performance characteristics, compatibility, and security of TASR. We have evaluated TASR against the SPEC 2006 benchmark and show that it offers significant protection with low performance overhead. The average runtime overhead incurred by TASR on SPEC 2006 is 2.1% with a maximum overhead of 10.1%. Thus, TASR's overhead is well within the commonly recommended threshold of 10% overhead for practical memory corruption defenses [44].

TASR is subject to certain limitations. First, it is designed to protect precompiled binary applications rather than interpreted code, and as such, attacks such as JIT-ROP [42] that apply to scripting engines are not in scope. Second, TASR cannot automatically handle code that is not compliant with the C Standard in certain ways. Specifically, upcasting any other pointer type into a function pointer prevents necessary code annotations during the compilation process. Third, use of a custom memory allocator requires the manual addition of the allocator signature into the compilation process in order to properly convey necessary information between compilation stages. Related to this, memory allocations must make use of the `sizeof()` operator when allocating memory that includes function pointers. Fourth, TASR does not protect against data-only attacks or attacks that use relative addressing, and as such, partial pointer overwrite attacks remain possible without the incorporation of additional fine-grained ASLR techniques. These limitations are described in more detail in Sections 3 and 4.

Our contributions are summarized as follows:

- We design a complete prototype that continuously randomizes memory layout in a manner that is synchronized with attacker interactions in order to thwart memory disclosure attacks (direct or indirect) that are commonly used to bypass existing defensive techniques.
- We implement a prototype for applications written in C on x86_64 Linux systems. We modify GCC and the dynamic linker to support rerandomization, and develop a kernel component and userspace module to control and perform it.
- We evaluate TASR using the SPEC 2006 benchmark and show that it incurs a low performance overhead. We illustrate that by tying output system calls to rerandomization events, we can minimize the overhead of TASR while providing the desired security guarantees.

We begin in Section 2 by reviewing existing defenses and their weaknesses with respect to memory disclosures. Section 3 details the threat model under which TASR operates. The design and implementation of our prototype make up Section 4. We evaluate the performance and security of our prototype in Section 5. Finally, we conclude in Section 6.

[1]Pronounced "taser"

2. BACKGROUND AND RELATED WORK

2.1 Randomization-Based Defenses

In 2001, the PaX team released Address Space Layout Randomization which took the form of a Linux kernel patch [35]. It applies a one-time randomization to various memory components of a process at load time. With the advent of sophisticated exploitation techniques such as ROP in which small code snippets are chained together to achieve a malicious purpose, finer-grained forms of ASLR have been proposed in the literature. For example, ASLP [27] randomizes the location of functions within libraries (a.k.a. medium-grained ASLR), and Binary Stirring [46] applies randomization at the basic block level (a.k.a. fine-grained ASLR). Oxymoron [7] is another technique that facilitates fine-grained randomization while allowing code sharing among processes.

Randomization can also be applied to the content of memory. For example, the multicompiler technique [24] diversifies the content of memory by adding random no-operation (NOP) instructions into the code at compile-time, among other protections. ILR [21] is another example that implement in-place code rewriting of binaries. Techniques such as multicompiler and ILR can achieve sufficiently high entropy to make traditional brute-force attacks [41] hard to implement without unacceptably high numbers of crashes.

The challenge in these techniques, however, arises in the frequency of randomization. Multicompiler randomizes the code at compile-time, and ASLR, ASLP, Binary Stirring, Oxymoron, and ILR randomize the code at load-time. Although the latter case ensures that the code or layout will be different every time that the process is run, the code and layout will then stay the same *during* the entire execution of the process which can be days, weeks, months, or longer in the case of some server programs.

2.2 Enforcement-Based Defenses

Numerous enforcement-based memory defenses have also been proposed in the literature. Complete memory safety techniques such as the SoftBound technique with its CETS extension [31] incur large runtime overhead (up to 4x slowdown). "Fat pointer" techniques such Cyclone [25] have also been proposed to provide spatial pointer safety, but are not compatible with existing C codebases. Other efforts such as Cling [4] and AddressSanitizer [39] only provide temporal pointer safety to prevent use-after-free attacks.

Control flow integrity (CFI) [3] techniques are another class of enforcement-based defenses. They enforce a compile-time extracted control flow graph (CFG) at runtime to prevent control hijacking attacks. Unfortunately, precise CFI enforcement has not yet shown itself to be practical [3]. As a result, weaker forms of CFI have been implemented in CCFIR [47] and bin-CFI [48], but have also been shown to be vulnerable to carefully crafted control hijacking attacks [18]. Other, finer-grained forms of CFI have also been proposed recently among which are: Opaque CFI [29] and Forward-Edge CFI [45].

2.3 Memory Disclosure

A major assumption in the existing layout and code randomization techniques is that the layout of memory remains unknown to the attacker. As many attacks have illustrated, however, memory content can leak directly or indirectly [22]. In the direct form, memory content is sent as an output of the process because of a memory disclosure vulnerability. For example, a *buffer overread* vulnerability can be used to leak memory content [43]. Moreover, the same vulnerability may be used repeatedly to leak large parts of memory, a technique that has been used to successfully bypass fine-grained ASLR [13] and Oxymoron [13]. In the indirect form, timing or fault analysis attacks are used to remotely leak the contents of memory. These attacks have been shown to be effective in bypassing one-time randomization techniques [38], and are effective even against unknown binaries [9]. In its indirect form, the attack can cover large regions of memory, and is not limited to areas adjacent to an overflowed buffer. Moreover, these attacks do not require a separate memory leakage vulnerability; that is, a single buffer overflow vulnerability can be used for the dual purposes of leaking the content of memory and then hijacking control [38]. The indirect information leakage attack [16] was also used to bypass an enforcement-based memory corruption defense known as code pointer integrity [28], and even recent generalized techniques such as Counterfeit Object-Oriented Programming [37] make use of information leakage to carry out an otherwise low-requirement attack. A promising new defense, Readactor [11], combines execute-only permissions with a randomized indirection layer to resist memory disclosure attacks. It provides much greater resistance to memory disclosure attacks than previous defenses, but maintains a one-time only randomization strategy.

TASR solves these memory disclosure problems by rerandomizing the layout of memory at every opportunity that an attacker has to obtain such information.

2.4 Previous Rerandomization Efforts

Memory rerandomization has been proposed [17] or mentioned in the literature [9, 13, 6]. To the best of our knowledge, the work by Giuffrida *et al.* [17] is the only one that provides an actual implementation, applying live rerandomization in the Minix 3 microkernel. This technique employs a wall clock timing-based rerandomization model. There are two drawbacks to such a timing-based approach. First, it imposes unnecessary overhead when the system is currently performing operations which permit no opportunity for attack. Second, and more importantly, even very frequent rerandomization may not provide sufficient security. For example, the technique proposed by Giuffrida *et al.* can rerandomize as fast as once a second, but we note that one second is sufficient to execute an attack even for remote attackers that require multiple network round trips: the attacker can leak memory content and send the control flow hijacking payload, all within the interval between two randomizations. Other discussions of possible rerandomization in the literature also focus on timing-based rerandomization. For example, Davi *et al.* [13] note that "However, the adversary could exploit the (small) time frame between the subsequent randomization to launch the attack", and Backes *et al.* [6] note that "an attacker can potentially abuse this long time window to perform a JIT-ROP attack". The novelty in TASR is tying rerandomization to output in order to synchronize it with potential observation points for an attacker, achieving efficiency while providing the desired security guarantees.

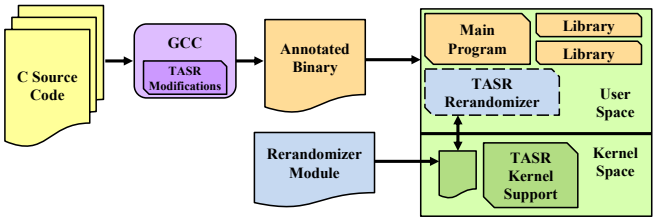

Figure 1: TASR Architecture

3. THREAT MODEL

In our threat model, an attacker has remote or console access to the application and can interact with it via standard system interfaces but is not explicitly authorized to directly access the memory of the application (e.g., via shared memory). However, the application is assumed to contain one or more memory corruption vulnerabilities that unintentionally allow the attacker to corrupt and read memory. Since memory corruption vulnerabilities are known to be strong for modifying or leaking memory content [38], we assume that the attacker exploits these vulnerabilities to read or write arbitrary memory pages. Thus, the attacker can take advantage of memory disclosure vulnerabilities that allow the layout of memory to be leaked in its entirety, in addition to vulnerabilities that enable the corruption of pointers, for the purposes of hijacking control flow. The attacker cannot, however, inject or modify code. This can be, for example, because W⊕X or Data Execution Prevention (DEP) has been enabled on the system. As a result, the attacker must resort to code reuse attacks. We further assume that the operating system, dynamic linker, and hardware devices are trusted. Our threat model is consistent with related literature on memory corruption [44] and code reuse attacks [9].

We also note that the TASR prototype is designed to protect precompiled application binaries. Applications making use of interpreted code, such as in script execution engines, are capable of leaking memory content to uninstrumented code in the same process address space without traversing a system call boundary. As such, attacks such as JIT-ROP [42] are out of scope for this work.

4. DESIGN AND IMPLEMENTATION

Many code reuse attacks begin with the exploitation of a memory disclosure vulnerability that reveals the location of an existing piece of executable code. TASR is designed to thwart these attacks. It repeatedly rerandomizes the location of all executable code at runtime, and does so at every opportunity an attacker has for observing the system.

This core functionality requirement is augmented by the practical considerations of ensuring low performance overhead, and avoiding the imposition of overly burdensome practices upon software developers. A recent survey suggests that security techniques must impose low overhead in order to be practical [44], and similarly, requiring manual code annotations or limiting existing standard coding practices (e.g., unions) is unscalable and unlikely to gain widespread adoption.

TASR's development was guided accordingly, and is described in the remainder of this section. We begin with the high-level direction and core architecture of TASR as a whole, and then proceed into the three major individual components.

4.1 High-Level Design and Core Architecture

We chose to implement TASR as a prototype for the x86_64 architecture running Linux, applied to programs written in the C language. The security and functionality requirements discussed above may be applied to a variety of different implementations, but this particular combination was chosen due to its general suitability for both experimental research projects and production systems, its robust and open-source toolchain and ecosystem, and its familiarity to developers. Additionally, C programs are particularly notorious for enabling memory corruption attacks, and are a common target for the type of code reuse attack that we aim to prevent.

We identified three high-level tasks for achieving effective rerandomization of a given program. First, we need to gather enough information on the code to be able to rerandomize it without breaking it. Second, it is necessary to select appropriate rerandomization points that deny an attacker any opportunity for a successful attack. Third, we require the rerandomization component itself.

Ideally, TASR could be applied to any arbitrary precompiled binary. Unfortunately, as is often the case, the opacity of compiled code renders it extraordinarily difficult to collect sufficient information about the program structure to apply the technique, using only static offline analysis. Dynamic analysis at runtime offers greater opportunities, but also imposes higher overhead. At the opposite end of the spectrum, a requirement for developers to write their code according to predefined and strict standards would render analysis trivial, but would also violate our requirement of minimal manual source code changes on the part of the developer.

TASR works between the two extremes: during compilation without manual modifications, so it is a *source compatible* technique in the taxonomy developed by Szekeres *et al.* [44]. Two opportunities are afforded to us during the compilation process. First, we can analyze the source code and extract necessary pointer information, carrying it along through the multi-stage compilation process as debugging information. Second, we can minimize the amount of debugging information for performance reasons. The rerandomization stage itself must be triggered during runtime, but it need not be integrated into the program as native code, and is not in the case of TASR.

We chose rerandomization points by reasoning about the threat model and following it to its natural conclusion. Our threat model assumes that an unknown attacker has knowledge of, and access to, both an arbitrary memory disclosure vulnerability and a control flow redirection vulnerability, in the targeted program. We assume that we cannot detect the use of either vulnerability, and are likely entirely unaware of their presence. Therefore, we must assume that knowledge of the program's memory layout may be undetectably leaked in any **output** from the program, and that control flow may be redirected at any point where the program acts upon external **input**.

Although we cannot identify the exact points at which the attacker actually exercises either vulnerability, we can identify every opportunity at which it is possible to do so. Specifically, an attacker can redirect control flow on the first input made to the program following the data leakage from

an output. Put more simply, the minimum interval to carry out an attack is the time between the most recent output and the following input. If rerandomization occurs at each such interval, there is never an opportunity to use knowledge of the previous memory state.

TASR uses a kernel component to invoke rerandomization at these points. Because the kernel handles all external I/O calls, it is in the best position to track I/O events across the entire process, including events in multiple threads. It would also have been possible to invoke rerandomization via code injected during the compilation process, but that approach would have required extra logic to handle the tracking of I/O ordering.

Finally, the actual memory rerandomization step has components that reside in both kernelspace and userspace. Although we initially considered both kernel-only and userspace-only approaches to rerandomization, we determined that a hybrid approach offered the best tradeoff between security, performance, and ease of implementation. The kernel selects the new random locations to which the code segments will be moved, carries out the actual movement, and injects a temporary userspace component into the memory space of the process being rerandomized. This userspace component is logically separate from the target process, but is considered to *be* the process for the duration of its existence, and it has full memory access thereby. All rerandomization tasks aside from the code movement are carried out by this userspace component, which the kernel then removes before returning control to the original program control flow.

The overall architecture of TASR is depicted in Figure 1. A compiler component, implemented as a customized version of GCC, compiles programs in a form suitable for rerandomization through TASR. The kernel component controls the timing of rerandomization and handles certain other bookkeeping tasks. An injected userspace component performs the bulk of the rerandomization in a way that is transparent to the original program. Detail on each component follows in the subsequent sections.

4.2 Compilation

The first two required steps in code rerandomization are to produce code that is capable of being easily rerandomized, and then to have sufficient information about that code to carry out the actual rerandomization step. Fortunately, the first step is not only a solved problem, but has been regularly applied to production code for many years in the form of PIE/PIC compiler options (Position Independent Executable/Code) for UNIX-like systems, and the equivalents in other major operating system environments. These compiler options are used to enable ASLR, and any program that supports ASLR is randomizable at least once and thus well-poised for future rerandomization. The second step, having sufficient information about that code to enable rerandomization, is more complicated.

At the time of rerandomization, all references to the code (i.e. function pointers and return addresses) must be updated to point at the new location of that code. To be updated, those references must be fully tracked throughout the lifetime of the program such that they can be identified and updated at any time. Such tracking must be precise and sound. Missed references will result in segmentation violations and thus program crashes, whereas falsely-identified references will result in corrupted data.

```
 1 int main() {
 2     int x = 20, y;
 3     uintptr_t x_loc, y_loc;
 4     int *x_ptr;
 5     x_loc = (uintptr_t)&x;
 6     y_loc = (uintptr_t)&y;
 7     x_ptr = (int *)x_loc;
 8     printf("x and y are %" PRIuPTR
 9            " bytes apart, and x = %d.\n",
10            (x_loc - y_loc), *x_ptr);
11     return(0);
12 }
```

Figure 2: Code with valid (ab)use of data pointers.

Code references come in two flavors. The first type is generated implicitly, and is comprised of inter-code references, intra-code references, and return addresses on the stack. None are explicitly declared in the C language, and are only accessible by taking advantage of architecture-specific knowledge of code and data layout. The second type is declared explicitly as a function pointer, as in this example: `int (*fptr)(int, char *);`. These references may be manipulated, set, and explicitly invoked at will by the programmer. So long as both reference types can be tracked and updated when needed, rerandomization should be possible.

TASR is intended to protect against code reuse attacks, and thus our focus is on code location rerandomization. The question naturally arises: should program code alone be moved, or should program data (possibly including dynamically allocated data segments) also be moved? We elected to move code only for two primary reasons: performance, and the difficulties in precisely tracking data reference locations. We discuss the security implications of this design choice in Section 5.

For performance, we noted that the incidence of references to code was extremely low compared to the incidence of references to data. Code references typically number in the hundreds, whereas it is common for a large program to have many thousands of data pointers, and millions are not unusual. Since each reference must be updated at each rerandomization point, this can impose significant performance overhead.

According to the C standard, one cannot cast *into* a function pointer from any other data type. This is of particular importance to us, because it ensures that a user-defined code reference cannot exist outside of a function pointer variable. Unfortunately, no corresponding restraint is placed upon data references, which makes ambiguous data references possible.

Consider the example in Figure 2. The integer variable `x_loc` is first used to set an integer pointer to point at a specific integer, and then used to calculate a byte offset. If rerandomization is invoked at line 6, we have the choice of treating `x_loc` as a data pointer (because we note that it was assigned from a data pointer), or as an integer (which is its native type). If `x_loc` is treated as a data pointer, the `printf` statement's calculation of `x_loc - y_loc` yields an incorrect result according to the original programmer intent. If `x_loc` is treated as an integer, line 7 yields an incorrect result because `x_loc` now points to an outdated location. There is no solution to this issue, short of requiring explicit code annotations or forbidding the use of a valid and common C construction. Neither solution is acceptable according to our initial requirements. Movement of code only avoids the need for either of these two undesirable options.

References to code can be divided into three major categories:

1. References to code residing within the same compilation unit.
2. References to code residing in other compilation units.
3. References to code that are assigned at runtime, of which there are three subtypes: (a) global variables, (b) local variables and return addresses residing on the stack, and (c) dynamic allocations.

Code references from category 1 require no special action beyond compilation as PIE/PIC, using standard compiler and linker flags. Position-independent code uses relative offsets to access code within the same unit, rather than absolute references. Because each code segment moves as a block, relative offsets do not change and there are no references that require updating.

Category 2 references similarly require no special action beyond standard compilation and linking options. References to code in other compilation units are not resolvable at the time of compilation, and thus are not assigned absolute addresses. Instead, references are made by relative offset into the Global Offset Table (GOT), a data structure present in each compilation unit. Thereafter, only one well-defined reference, residing at a known location in the GOT, need be updated at each rerandomization.

Code references of subtypes 3(a) and 3(b) from category 3 can be identified at compilation time and pose a special problem only in the context of unions. Global variables reside at permanent known locations, and items on the stack exist in calculable locations. In the case of unions, a single data location may contain a function pointer or may contain some other data type, and its contents are non-deterministic at compilation time. Therefore, during the compilation phase, we use the technique proposed by Rafkind, *et al.* [36] to add code to create *tagged* unions wherever such unions may contain a function pointer. This incorporates an extra data field to label the present type contained in the union, and at each union assignment, the label is updated to mark it as containing a function pointer, or some other data type.

Finally, code references of subtype 3(c) from category 3 are non-determinable at compilation time. Therefore, a runtime tracking component must be inserted into any program that contains dynamically allocated function pointers. This component is added automatically during compilation by incorporating the equivalent of a small library, through which all dynamic allocations are routed. Instructions are automatically generated to record the dynamically allocated locations of function pointers in a table established at runtime. Entries are added and subtracted as they are allocated and deallocated.

This solution comes at a price. Dynamic allocations must make use of the `sizeof()` operator in order for the compiler to understand what types are present. The statement `void *mem = malloc(10 * sizeof(struct some_structure))` is interpreted as an allocation of a block of memory containing ten elements of that particular structure, from which the locations of function pointers are calculated. In contrast, `void *mem = malloc(800)` has no such metadata present, and nothing can be inferred about the location of potential function pointers within that block. Fortunately, in our evaluation, we show that memory allocations without making use of `sizeof()` are rare in practice for major programs (none in the SPEC2006 suite; see Section 5), and are typically only used for `char` arrays, and can thus be assumed to contain no function pointers in this context. If such allocations are discovered, the TASR compiler component issues a visible warning and treats any such allocation as a character array not containing function pointers.

The preceding code generation steps ensure that the program can be rerandomized, so long as precise location information for non-dynamic code references is made available. Such tracking requires location information at every single program instruction, including the exact memory addresses (calculable through an offset) and hardware registers in which the reference resides. As previously discussed, missing even a single pointer sets up a segmentation violation crash situation, and false identifications corrupt data.

GCC has built-in compiler options that begin to address this problem: the `-fvar-tracking` and `-fvar-tracking-assignments` debugging options. These options are intended to compute the locations at which variables are stored for each instruction, and that information is carried through the compilation process and emitted as debugging output at the end. The implementation of these tracking options was not entirely complete in GCC 4.8.2, which we used as a base. It did not properly track global variables, variables that existed in multiple simultaneous locations, certain transient locations where temporary variables were being used in a different stage of the compilation process, around transition points when passing parameters into functions, in certain structure operations, with arrays of structures, and in certain other miscellaneous corner cases. We modified GCC to track all of these cases and emit the appropriate debugging information.

In standard compilation, objects in the data segment are usually referenced by relative addressing from the text segment. This would normally represent a problem for TASR, which repeatedly moves the text segment while leaving behind the data segment at its original load-time position, because all relative references between the two sections are invalidated after the first move. To handle this problem, we modified GCC so that it treats all data segment object references as if they were externally visible global references residing in different compilation units. We then rely upon GCC's normal handling of such references, which is to route them through the GOT. This converts all relative references to the data section into relative references to the GOT, and because TASR moves the GOT relative to the text segment, all relative references remain valid after each runtime rerandomization. As per references to code residing in other compilation units, only this one well-defined reference need be updated at each randomization. This results in an extra level of indirection for all statically allocated variables; however, it contributed no significant penalty to CPU overhead or program memory usage in our tests.

The final product of this process is a standard ELF object file, including a debugging section in the DWARF format [15]. It also contains one extra program segment: the "TASR_GOT". As previously mentioned, the Global Offset Table is a data section that contains references to code, and is accessed via relative offset from the code. This data section is an exception to the code-only movement, and is itself treated as if it were part of the code segment that references

it. It is safe to move this data segment, since user code cannot normally gain direct reference to the GOT without taking particularly convoluted actions, for which we have been unable to identify any practical use, and nor have we detected any instances of this happening in our testing. The extra segment and name "TASR_GOT" also serves a more utilitarian purpose: it immediately identifies an ELF file as being TASR-enabled.

4.3 Invocation and Kernel Support

TASR modifications to the Linux kernel include support for TASR process start-up, appropriate triggers for rerandomization, selection of new movement locations within the address space, and maintenance of certain kernel-stored code references. To do this, the kernel keeps a small amount of per-process TASR information (proportional to the number of TASR-enabled modules in the program), monitors I/O related system calls, and initializes userspace code to handle the actual rerandomization. The kernel itself does not perform the rerandomization according to the principle of least privileges; the rerandomization functionality has no need for elevated privilege and its actions are easily handled in userspace without kernel involvement beyond the initial setup.

In Linux, all communication that crosses the process address space boundary must route through the kernel by means of a system call. Such communication occurs through file descriptors (for both standard filesystem files and "special files" like named pipes) through system calls in the `read()` and `write()` families. Therefore, tracking those system calls allows us to appropriately time rerandomization. The system calls used for I/O are listed in Table 1. In the remainder of this text, references to `read()` and `write()` are intended to represent the entire family of calls unless otherwise noted. Also note the inclusion of `fork()` and `vfork()` system calls in the "Input" column. Although not technically "input" system calls, treating them as such ensures that two correlated processes have different memory layouts at the time of their split.

Therefore, the rerandomization strategy is as follows: rerandomize before any `read()` that follows one or more `write()` calls. As previously discussed, this strategy is best imposed by the kernel. Not only is the kernel involved in each system call, but also it is in a position to correlate `read()` and `write()` calls between processes in a process group, without which multithreaded programs would go unrandomized over multiple I/O cycles in split threads. The rerandomization itself, also as previously discussed, takes place in userspace. Although we initially considered kernel space rerandomization, security best practices dictate that we spend as little time as possible in kernel mode. Therefore, we developed a method of code segment injection: rerandomization code can be injected into the address space and control flow transferred to that code. This also allows the use of userspace libraries without the necessity of porting them to the kernel and the risk of introducing additional vulnerabilities thereby.

The initialization of a TASR process is recognized in the kernel by the presence of the "TASR_GOT" program segment in the ELF file, as generated during compilation. The process is loaded normally with a small amount of TASR data also associated with the process. Execution commences normally.

Input		Output	
#	syscall	#	syscall
0	`read()`	1	`write()`
17	`pread64()`	18	`pwrite64()`
19	`readv()`	20	`writev()`
45	`recvfrom()`	44	`sendto()`
47	`recvfrom()`	46	`sendmsg()`
243	`mq_timedreceive()`	242	`mq_timedsend()`
295	`preadv()`	296	`pwritev()`
299	`recvmmsg()`	307	`sendmmsg()`
Input-like			
57	`fork()`		
58	`vfork()`		

Table 1: System call numbers and names for input and output on the x86_64 architecture.

Aside from startup and shutdown (including calls to `exec()`, `fork()`, and the `exit()` family), kernel behavior with regard to a TASR program differs in only two regards. First, any use of a `write()` family system call (see Table 1) toggles a flag in the process (or process group) to indicate that output was produced. Second, any use of a `read()` family system call checks that output flag, triggers a rerandomization if it is set, and then clears the flag again.

Actual rerandomization works as follows. First, all virtual memory areas are checked in order to determine if they should be rerandomized. The addition of new areas is relatively rare: standard linked libraries are examined at the time of the first rerandomization and the list subsequently changes only when new libraries are dynamically added or subtracted. For each TASR-enabled object, a new random address in memory is generated. This addressing information is then placed into a userspace memory segment of appropriate size, along with a copy of the original register set and the addresses used for dynamic runtime tracking of code references.

The kernel then injects a component called the "pointer updater" into the process. This component consists of the virtual memory areas of what could normally be considered a separate process, along with certain other information that allows it to be injected, in toto, into any other address space. Control is transferred to this injected component with the address of the aforementioned userspace memory segment as an initial parameter. Specifics of the pointer updater component are discussed in subsection 4.4, and do not involve the kernel between setup and withdrawal from userspace. When the pointer updater component completes its task and exits, the kernel withdraws it from userspace, tidies up any memory areas that were not part of the original process, updates the register set of the original process if required, and performs the actual move of the memory segments in question. This movement is efficiently made by updating the process page tables rather than copying the contents of memory from one location to another. Control is then returned to the original program. The rerandomization is thus seamless to the original process.

The kernel must also maintain the location of any set signal handling functions, a task easily accomplished since the kernel is the one to choose new addresses for all TASR-enabled code. Should correlation between processes be required in the context of multi-threaded programs sharing address space, it also tracks I/O calls across an entire process group.

4.4 Rerandomization Process

The pointer updater bears most hallmarks of a standalone process, except that it runs in the address space of a TASR process that is currently being rerandomized, rather than in its own. It would ideally be a self-contained segment within the kernel, but has been initially implemented in the prototype as a standalone process for reasons of developmental convenience. It is run once at startup for initialization purposes, then withdrawn into the kernel.

Rerandomization requires that code references in the three major categories enumerated in Section 4.2 be updated. Code references most likely exist at specific addresses in memory, but may also exist in the active register set if it so happens that a code pointer is "in use" at the time of rerandomization. The addresses to which code pointers must be updated are calculable from the information provided by the kernel. Each pointer location is examined in turn. When the pointer contains a valid code address (it may not, if not yet initialized), its new value is computed and updated.

The locations of dynamically allocated code pointers reside in a simple list, to which the pointer updater has a reference by means of the kernel-passed parameter. Updating these pointers is merely a case of iterating through the list and applying the transformation described above. Other code references are more complicated.

As a result of changes discussed in subsection 4.2, the DWARF debugging information contains a complete reference to all global and local variables. The data is assembled into a data structure known as an "interval tree" which allows fast variable lookup based on the program instruction pointer. This data structure is used in conjunction with stack unwinding to step back through each stack frame of the process and determine what variables were in use during each frame. For each stack frame, the instruction pointer is used to query the interval tree and return a complete list of code pointers in use, and the memory addresses (or registers) in which they are located at that time. Each pointer is updated in turn. Global variables and other global references, which reside at static locations, are also queried via the DWARF interval tree but do not depend on the current instruction pointer.

After all updates are made, the pointer updater returns control to the kernel and shortly thereafter ceases to exist in the address space of the program being rerandomized. No state is maintained between runs of the pointer updater.

As a userspace component, we were able to make use of a large body of existing code, most notably `libunwind` [30] for the purpose of stack unwinding. This library is not particularly well-suited to this task, and required modification to enable desired functionality. We speculate that moderate performance improvement would be possible with the use of a more efficient library. Similarly, we suspect that there are opportunities for significant performance improvements in DWARF parsing and variable tracking. Our current implementation of the DWARF interval tree includes all variables rather than only code pointers, and must be assembled anew on each run of the pointer updater. Significant performance improvements could likely be achieved by optimization of DWARF parsing and pre-generation of the needed interval tree as a one-time compilation step. However, performance results (see Section 5) are initially favorable even without these optimizations, and we have instead concentrated our efforts on other areas.

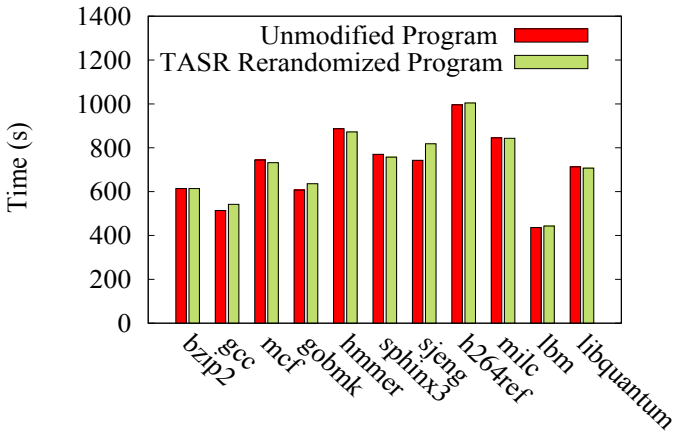

Figure 3: Runtime overhead for TASR rerandomization triggering at input/output system calls for the SPEC CPU2006 Benchmark

4.5 Limitations

TASR is subject to certain caveats and limitations beyond those that have already been discussed. The use of inline assembly in a C program cannot be tracked by our current methods and would require manual annotation of the assembly snippets in order to allow DWARF information to be generated. Should the author of a process manipulate code pointers in a block of inline assembly, such manipulation will not show up in DWARF tracking information and it becomes possible that a code pointer will not be updated during rerandomization. Naturally, the inability to update even a single code pointer will likely result in eventual program crash via segmentation violation when attempting to run code at a location not presently allocated.

5. EVALUATION

The goals of our evaluation are to determine the performance overhead of executing TASR-enabled binaries, examine the frequency of coding practices in common applications that would require manual modification in order to be compatible with TASR, and to analyze the technique's effectiveness against attacks that leverage memory disclosure vulnerabilities in order to hijack control flow. These items are addressed in the following subsections.

5.1 Performance

The CPU and memory overhead of TASR were evaluated using the C programs in the SPEC CPU2006 benchmark, with the exception of `perlbench`, which was excluded due to TASR not currently supporting interpreted environments as per Section 3 (see Section 5.2 for further details). We acknowledge that the SPEC benchmark has certain limitations in evaluating TASR's overhead. Since SPEC CPU is a compute-intensive benchmark, it is primarily useful in evaluating the runtime performance impact of the allocator instrumentation in both CPU and memory overhead, but does not capture the full impact of rerandomization overhead in an I/O-intensive program (such as a webserver). Ideally, TASR should also be evaluated on an appropriate I/O-intensive benchmark, but such benchmarks are unfortunately not as widely used or available as SPEC.

All experiments were conducted on a Debian 7 machine running Linux kernel 3.2.65 on a 4-Core Intel Xeon 2.66 GHz processor with 8 GB of RAM. The performance of TASR-enabled programs is compared against the performance of the same set of programs compiled using an unmodified GCC 4.8.2 and using the default ASLR base address randomization but without runtime rerandomization. The baseline programs were compiled with GCC's unmodified DWARFv4 debugging information (`-gdwarf-4` and `-fvar-tracking` flags) while the TASR-enabled set was compiled with the additional debugging information and runtime instrumentation code required for rerandomization. Both the baseline and the TASR-enabled binaries were compiled using the -Og flag, which performs optimizations that preserve debugging information.

As shown in Figure 3, the CPU overhead ranged from negligible in more than half the cases to 10.1% in the worst case, with an average overhead of 2.1%.

Note that in a few cases the difference was sufficiently insignificant such that TASR binaries actually reported marginally faster runtimes than unmodified binaries, within the limits of expected variation. This is normal and, in fact, expected due to layout randomization. A similar phenomenon has been observed in the related work [11, 12]. Overhead in comparison to the total number of I/O pairings during the program run, average number of function pointers that needed to be updated at rerandomization time, and the average depth of the stack at rerandomization time are detailed in Table 2. Virtual memory overhead ranged from negligible in the best case up to 3.5MB in the worst case, with an average overhead of 1MB. The increase in memory consumption is primarily attributable to the additional DWARF debugging information that is added to binaries for tracking function pointer locations and unwinding the stack. The storage of debugging information was not optimized in any way and the memory overhead could likely be significantly reduced by performing such optimizations.

The very low overhead of TASR is because of its synchronization with the I/O system calls. Note that the expensive context switching cost (from userspace to kernelspace and back) is already paid by the system call when rerandomization is in-sync with read/write calls.

5.2 Compatibility

TASR successfully compiled and rerandomized all tested programs with the exception of `perlbench`. The `perlbench` component interprets code at runtime and TASR currently does not have the mechanism for adding debug information to track interpreted or runtime-generated code. Similar mechanisms that were added to GCC would need to be added to the Perl interpreter in order to generate such debug information at runtime. Moreover, we observed that `perlbench` violated the C standard by treating function pointers as data pointers in some areas, which prevents accurate compilation under TASR without manual modifications. We thus did not evaluate `perlbench` under TASR.

As discussed in Section 4.2, there are certain cases where TASR can miss pointers to code and thus would fail to update the pointers at rerandomization time. Specifically, this occurs when function pointers are stored as other types but later used as function pointers, or where dynamic allocations of function pointers cannot be automatically recognized by TASR due to custom allocators or the lack of any type information passed to `malloc()`. These cases require manual modifications of the source code such that function pointers are indeed declared as function pointer types. There are four programming practices that could require manual modification in order to be compatible with TASR: function pointer casting, allocations without type information, union type punning with function pointers, and custom memory allocators.

Function pointer casting: TASR assumes strict compliance with the C standard's rules on conversions involving function pointers. Specifically, a program should not cast a non-function pointer into a function pointer [23]; thus TASR only modifies pointers that are declared as pointing to a function. Pointers that are declared as pointing to a non-function and later converted to pointers to functions may cause the program to fail. Outside of a few specific instances (interpreter implementations and usage of POSIX's `dlsym`) these conversions are not common. We analyzed the frequency of casting between function pointers and object pointers using GCC's `-pedantic` flag. This flag enables all warnings demanded by strict ISO C compliance. Several of these warnings indicate the presence of a cast between function pointers and object pointers. The total number of function pointer casts found are shown in the two rightmost columns of Table 2. Note that not all function pointer casts enumerated in those columns actually cause a problem for TASR. Typically only those cases where an object pointer is upcast to a function pointer prove to be problematic in practice. In these cases, the mistyped function pointer would not be updated during rerandomization and thus would contain a stale address that would break the program if the function pointer were subsequently called. While there were some function pointer casts in the SPEC benchmark programs, we found no actual problematic cases (i.e. upcasting) during our evaluation.

Union type punning: TASR's union tagging requires that whenever a union member is accessed, that member must be the same as the member last used to store a value into the union. Occasionally programmers will intentionally access a union member that does not correspond to the last assigned union member to reinterpret the stored value as a different type (a process known as *type-punning*). In practice, this does not pose a problem to TASR as the number of union members of type pointer to function is low as shown in the two rightmost columns of Table 2. While we did find some cases of unions containing function pointers, we did not find any actual instances of type-punning in our test programs.

Malloc without type: TASR tracks function pointers located on the heap by annotating all allocations with type information. This type information is inferred via a syntactic analysis on the call site of a heap allocation. The syntactic analysis examines the arguments passed to the allocator, looking for a `sizeof()`. If a `sizeof()` is found, the type inside the `sizeof()` will be used to annotate allocations made from that site. If no `sizeof()` is found, it examines the type of the pointer to which the newly allocated space is assigned. If the pointer is of type pointer to void, our analysis will display a warning as it is unable to determine the type associated with the allocation. We did not find any such instance in the SPEC benchmark programs.

Custom allocators: Many large programs use custom allocators and these need to be instrumented for TASR to

Program	CPU Overhead	Memory Overhead (KB)	I/O Pairs	Mean Code Ptrs	Mean Stack Depth	Function Ptr Casts	Unions Containing Function Ptrs
bzip2	∼0%	28	6	144	5	0	0
gcc	5.5%	2512	0	0	0	175	0
mcf	∼0%	84	2	147	6	0	0
gobmk	4.8%	3156	551	2019	11	0	3
hmmer	∼0%	264	2	223	10	0	0
sphinx3	∼0%	88	124	209	4	0	5
sjeng	10.1%	3436	2	169	6	0	0
h264ref	.8%	1002	36	212	6	0	21
milc	∼0%	48	2	167	11	0	1
lbm	1.6%	680	0	146	7	0	1
libquantum	∼0%	0	0	0	0	1	1

Table 2: Overhead in relation to frequency of I/O Pairs (rerandomizations), code pointers requiring updating, and depth of the stack at rerandomization time. The two rightmost columns show the frequency of function pointer casting and unions containing function pointers. Note that none of these cases were actually problematic and thus no manual modification was necessary.

track dynamic allocations. Currently, this requires one to determine the signatures of all custom allocators used by a program and to add the signature to TASR. This usually requires minimal manual effort as most programs tend to have only a few custom allocators. Of the SPEC C programs, only `gcc` and `perlbench` contain custom allocators and there are only 31 between them [10].

5.3 Security

5.3.1 Analysis

Traditional memory disclosures directly leak a memory address or part of an address via benign but vulnerable code. For example, a format string or buffer overread vulnerability such as Heartbleed [19] could be used by attackers to coerce the program to output return addresses or function pointers. Following the output, the attacker can use the learned addresses to craft a subsequent payload that redirects control flow to locations discovered via this leak. TASR triggers rerandomization prior to allowing the process to complete an input system call, thus preventing such attacks by rendering the addresses used in the payload stale by the time the program attempts to act upon them. In contrast to the case of standard ASLR where such attacks succeed with certainty, TASR effectively reverts the attacker's use of a known address to a mere guess of an address that succeeds with a probability corresponding to the baseline entropy of the targeted memory object. The guess will likely result in the process crashing. While DoS attacks of this variety are still possible under TASR, code execution attacks are prevented with high probability.

Another form of memory disclosure allows an attacker to indirectly leak memory addresses by guessing addresses and observing whether or not a process crashes. There are techniques that are more effective than simple brute force guessing: they perform partial overwrites of existing addresses and guess one byte of an address at a time rather than a full 64-bit address, reducing the expected number of guesses on 64-bit Linux from 2^{27} to 640 [9]. This piecewise guessing technique is commonly known as stack reading and was generalized for use in a form of code reuse attacks known as Blind ROP attacks [9]. These attacks rely upon the process automatically invoking `fork()` after the crash and with the same address layout. As TASR triggers rerandomization each time `fork()` is called, any successful guesses of a single byte of an address are immediately rendered stale at the next I/O pairing or failed guess, preventing the attacker from building upon piecewise guesses.

Similarly TASR prevents remote side-channel attacks (timing and fault analysis) such as those proposed by Seibert *et al.* [38] because they still rely on (side properties of) system output to leak memory content. TASR rerandomizes the memory after every output rendering information gained through these side-channels stale.

The recently proposed COOP attack [37] is also prevented by TASR because it requires the knowledge of the memory layout (at least for "the base addresses of a set of C++ modules"). COOP assumes the presence of a direct or indirect memory disclosure vulnerability. TASR prevents COOP by hindering an attacker's ability to gain this knowledge.

5.3.2 Nginx Memory Disclosure Attack

We evaluated TASR on a version of Nginx vulnerable to a stack-based buffer overflow [1] discovered in 2013. This vulnerability can be used to disclose the contents of the stack, including return addresses, by writing a guessed value to the stack and observing whether or not Nginx crashes, in an attack known as Blind ROP [9]. We used a version of the Braille [2] exploit tool optimized for conducting this attack against ASLR-enabled Nginx binaries.

We found that TASR broke this attack in the first of the six attack stages (disclosing the value of a return address). Braille's attack depends on being able to read a return address byte-by-byte over the course of several requests. However, when Nginx is run with TASR each request triggers a rerandomization due to either an I/O operation or a *fork()* system call. Due to these rerandomizations, when Braille finishes the first stage of its attack, it is left with a return address that does not point to the text section of any running Nginx process. Braille then proceeds to the second stage of its attack (finding both a stop gadget and the PLT). Braille uses the return address it read in the first stage as an origin and scans backwards looking for a stop gadget. A stop gadget is any gadget which causes the program to block, such as a *sleep()* system call. During our evaluation, as expected, this stage never terminated when run against a TASR-protected binary as the pointer Braille used as the origin of its scan does not point to the text section of any running Nginx process.

5.3.3 Limitations

An ideal rerandomization approach would rerandomize the entire address space, including the data segments. However, TASR leaves data segments in fixed locations after the initial load-time randomization has been applied by base ASLR. The current state of C programs makes it challenging to definitively distinguish pointers to data segments from non-pointer types. While the C standard places restrictions on casting between function pointer and non-function pointer types, there are no such restrictions in casting pointers to data to non-pointer types. It is quite common for programs to cast integer pointers into simple base integer types and vice versa. This results in the inability to identify at rerandomization time whether or not a variable contains a pointer or a data value, and thus makes the movement of data unreliable. Unfortunately, excluding data segments from rerandomization affords minimal additional protection from data-only modifications over base ASLR. An attacker that leaks the location of a data object will be able to access that object in its same location following the leak.

The ability to modify data additionally has implications for control flow hijacking attacks. Attackers that leak the *location* of a function pointer or return address may be able to indirectly use the code pointer at that location. Attackers can overwrite *pointers to code pointers* to point to the *location* of a leaked code pointer. Should the benign application later attempt to dereference and call the *pointer to code pointer*, control will be redirected to the leaked code pointer instead. While an attacker does not know which value to use for the code pointer itself since it has been subject to rerandomization, the layer of indirection makes it sufficient to simply know the *location* of the code pointer, not the value itself; this location is in the data segment which remains fixed.

It is important to note that this attack is far weaker than the ROP attacks. In particular, the attacker would not be able to easily chain ROP gadgets that point to arbitrary addresses as is currently possible. The attacker cannot use `ret` or direct/indirect `jmp` instructions to reach arbitrary gadgets since the attacker would not actually know the location of any gadgets themselves. The attacker is limited to using function pointers or return addresses that natively exist in the benign program at the time of attack and whose locations were leaked. Additionally, the attacker could only reach them via existing double indirection code (i.e. code that dereferences and calls a function pointer).

The TASR prototype is also constrained to protecting against leaks that operate at I/O boundaries. This excludes attacks from processes that already have shared access to memory given that addresses could be leaked without any system output from the vulnerable process. This would also limit TASR's effectiveness in environments where the attacker's code may be running in the the same address space as the vulnerable program, such as web browsers interpreting JavaScript. To protect these applications, the I/O boundary would need to be set as the transition between the user-supplied JavaScript that is to be interpreted and the vulnerable engine code that is being exploited. As mentioned earlier in this section, TASR does not currently support interpreted or runtime generated code but could be extended to do so using the same principles described here. We leave this component to future work.

Control flow hijacking attacks that do not rely upon memory disclosures naturally remain possible. One such attack is a partial overwrite attack [14] in which an attacker is able to overwrite part of an address in order to reach a target object that is fixed relatively to that address. Rerandomization does not impact the relative positioning of objects. Fine-grained ASLR techniques that randomize the relative positioning of objects [21, 46, 27] provide one promising approach that could complement TASR in order to protect against these attacks, and in fact TASR already makes use of whatever base ASLR technique that the system can support.

6. CONCLUSION

TASR is a technique that introduces the concept of "timeliness" to the broader area of code randomization in order to enable address space *re-randomization*. It builds upon the concepts introduced by ASLR and allows programs to be rerandomized at arbitrary points in their lifespan. By careful selection of those rerandomization points, the impact of memory disclosure vulnerabilities can be entirely mitigated as applied to code reuse attacks. Because attackers never have the opportunity to make use of the information disclosed, a successful code reuse attack cannot be carried out.

Application of TASR imposes a reasonable performance overhead (2.1% on average) over a wide variety of tested programs. Although not yet applicable to certain classes of program, it requires only recompilation for many existing codebases without requiring further manual changes.

Future work will focus on adding support for interpreted environments and additional custom memory allocators.

7. REFERENCES

[1] Cve-2013-2028. Online, 2013.

[2] Blind return oriented programming. Online, 2014.

[3] Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. Control-flow integrity. In *Proc. of ACM CCS* (2005).

[4] Akritidis, P. Cling: A memory allocator to mitigate dangling pointers. In *Proc. of USENIX Security* (2010).

[5] Anderson, J. P. Computer security technology planning study. volume 2. Tech. rep., DTIC Document, 1972.

[6] Backes, M., Holz, T., Kollenda, B., Koppe, P., Nürnberger, S., and Pewny, J. You can run but you can't read. In *Proc. of ACM CCS* (2014).

[7] Backes, M., and Nürnberger, S. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. *Proc. of USENIX Security* (2014).

[8] Barrantes, E. G., Ackley, D. H., Palmer, T. S., Stefanovic, D., and Zovi, D. D. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proc. of ACM CCS* (2003).

[9] Bittau, A., Belay, A., Mashtizadeh, A., Mazieres, D., and Boneh, D. Hacking blind. In *Proc. of IEEE S&P* (2014).

[10] Chen, X., Slowinska, A., and Bos, H. Membrush: A practical tool to detect custom memory allocators in c binaries. In *Proc. of WCRE* (2013).

[11] Crane, S., Liebchen, C., Homescu, A., Davi, L., Larsen, P., Sadeghi, A.-R., Brunthaler, S., and Franz, M. Readactor: Practical code randomization resilient to memory disclosure. In *IEEE S&P* (2015).

[12] Curtsinger, C., and Berger, E. D. Stabilizer: Statistically sound performance evaluation. In *Proc. of ASPLOS* (2013).

[13] Davi, L., Liebchen, C., Sadeghi, A.-R., Snow, K. Z., and Monrose, F. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. *Proc. of NDSS* (2015).

[14] Durden, T. Bypassing pax aslr protection, 2002.

[15] Eager, M. J. Introduction to the dwarf debugging format. *Group* (2007).

[16] Evans, I., Fingeret, S., Gonzalez, J., Otgonbaatar, U., Tang, T., Shrobe, H., Sidiroglou-Douskos, S., Rinard, M., and Okhravi, H. Missing the point(er): On the effectiveness of code pointer integrity. In *Proc. of IEEE S&P* (2015).

[17] Giuffrida, C., Kuijsten, A., and Tanenbaum, A. S. Enhanced operating system security through efficient and fine-grained address space randomization. In *Proc. of USENIX Security* (2012).

[18] Göktas, E., Athanasopoulos, E., Bos, H., and Portokalidis, G. Out of control: Overcoming control-flow integrity. In *Proc. of IEEE S&P* (2014).

[19] Heartbleed.com. The heartbleed bug. Online, 2014.

[20] Hiser, J., Nguyen, A., Co, M., Hall, M., and Davidson, J. Ilr: Where'd my gadgets go. In *Proc. of IEEE S&P* (2012).

[21] Hiser, J., Nguyen, A., Co, M., Hall, M., and Davidson, J. Ilr: Where'd my gadgets go. In *Proc. of IEEE S&P* (2012).

[22] Hobson, T., Okhravi, H., Bigelow, D., Rudd, R., and Streilein, W. On the challenges of effective movement. In *Proceedings of the First ACM Workshop on Moving Target Defense* (2014), pp. 41–50.

[23] ISO. *ISO/IEC 9899:2011 Information technology — Programming languages — C*. 2011.

[24] Jackson, T., Salamat, B., Homescu, A., Manivannan, K., Wagner, G., Gal, A., Brunthaler, S., Wimmer, C., and Franz, M. Compiler-generated software diversity. *Moving Target Defense* (2011), 77–98.

[25] Jim, T., Morrisett, J. G., Grossman, D., Hicks, M. W., Cheney, J., and Wang, Y. Cyclone: A safe dialect of c. In *USENIX* (2002).

[26] Kc, G. S., Keromytis, A. D., and Prevelakis, V. Countering code-injection attacks with instruction-set randomization. In *Proc. of ACM CCS* (2003).

[27] Kil, C., Jun, J., Bookholt, C., Xu, J., and Ning, P. Address space layout permutation (aslp). In *Proc. of ACSAC* (2006).

[28] Kuznetsov, V., Szekeres, L., Payer, M., Candea, G., Sekar, R., and Song, D. Code-pointer integrity.

[29] Mohan, V., Larsen, P., Brunthaler, S., Hamlen, K., and Franz, M. Opaque control-flow integrity. In *Proc. of NDSS* (2015).

[30] Mosberger, D. The libunwind project, 2014.

[31] Nagarakatte, S., Zhao, J., Martin, M. M., and Zdancewic, S. Softbound: Highly compatible and complete spatial memory safety for c. In *Proc. of PLDI* (2009).

[32] Nagarakatte, S., Zhao, J., Martin, M. M., and Zdancewic, S. Cets: Compiler enforced temporal safety for c. In *Proc. of ISMM* (2010).

[33] One, A. Smashing the stack for fun and profit. *Phrack magazine 7*, 49 (1996), 14–16.

[34] Parno, B., McCune, J. M., and Perrig, A. Bootstrapping trust in commodity computers. In *Proc. of IEEE S&P* (may 2010), pp. 414 –429.

[35] PaX. Pax address space layout randomization, 2003.

[36] Rafkind, J., Wick, A., Regehr, J., and Flatt, M. Precise garbage collection for c. In *Proc. of ISMM* (2009).

[37] Schuster, F., Tendyck, T., Liebchen, C., Davi, L., Sadeghi, A.-R., and Holz, T. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *Proc. of IEEE S&P* (2015).

[38] Seibert, J., Okhravi, H., and Soderstrom, E. Information leaks without memory disclosures: Remote side channel attacks on diversified code. In *Proc. of ACM CCS* (2014).

[39] Serebryany, K., Bruening, D., Potapenko, A., and Vyukov, D. Addresssanitizer: A fast address sanity checker. In *USENIX* (2012).

[40] Shacham, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proc. of ACM CCS* (2007).

[41] Shacham, H., Page, M., Pfaff, B., Goh, E.-J., Modadugu, N., and Boneh, D. On the effectiveness of address-space randomization. In *Proc. of ACM CCS* (2004).

[42] Snow, K. Z., Monrose, F., Davi, L., Dmitrienko, A., Liebchen, C., and Sadeghi, A.-R. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Proc. of IEEE S&P* (2013).

[43] Strackx, R., Younan, Y., Philippaerts, P., Piessens, F., Lachmund, S., and Walter, T. Breaking the memory secrecy assumption. In *Proc. of EuroSec'09* (2009), pp. 1–8.

[44] Szekeres, L., Payer, M., Wei, T., and Song, D. Sok: Eternal war in memory. In *Proc. of IEEE S&P* (2013).

[45] Tice, C., Roeder, T., Collingbourne, P., Checkoway, S., Erlingsson, Ú., Lozano, L., and Pike, G. Enforcing forward-edge control-flow integrity in gcc & llvm. In *Proc. of USENIX Security* (2014).

[46] Wartell, R., Mohan, V., Hamlen, K. W., and Lin, Z. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proc. of ACM CCS* (2012), pp. 157–168.

[47] Zhang, C., Wei, T., Chen, Z., Duan, L., Szekeres, L., McCamant, S., Song, D., and Zou, W. Practical control flow integrity and randomization for binary executables. In *Proc. of IEEE S&P* (2013).

[48] Zhang, M., and Sekar, R. Control flow integrity for cots binaries. In *Proc. of USENIX Security* (2013).