ASLR-Guard: Stopping Address Space Leakage for Code Reuse Attacks

Kangjie Lu, Chengyu Song, Byoungyoung Lee, Simon P. Chung,
Taesoo Kim, and Wenke Lee
School of Computer Science, Georgia Institute of Technology

ABSTRACT

A general prerequisite for a code reuse attack is that the attacker needs to locate code gadgets that perform the desired operations and then direct the control flow of a vulnerable application to those gadgets. Address Space Layout Randomization (ASLR) attempts to stop code reuse attacks by making the first part of the prerequisite unsatisfiable. However, research in recent years has shown that this protection is often defeated by commonly existing information leaks, which provides attackers clues about the whereabouts of certain code gadgets. In this paper, we present ASLR-Guard, a novel mechanism that completely prevents the leaks of code pointers, and render other information leaks (e.g., the ones of data pointers) useless in deriving code address. The main idea behind ASLR-Guard is to render leak of data pointer useless in deriving code address by separating code and data, provide a secure storage for code pointers, and encode the code pointers when they are treated as data. ASLR-Guard can either prevent code pointer leaks or render their leaks harmless. That is, ASLR-Guard makes it impossible to overwrite code pointers with values that point to or will hijack the control flow to a desired address when the code pointers are dereferenced. We have implemented a prototype of ASLR-Guard, including a compilation toolchain and a C/C++ runtime. Our evaluation results show that (1) ASLR-Guard supports normal operations correctly; (2) it completely stops code address leaks and can resist against recent sophisticated attacks; (3) it imposes almost no runtime overhead (< 1%) for C/C++ programs in the SPEC benchmark. Therefore, ASLR-Guard is very practical and can be applied to secure many applications.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

General Terms

Security

Keywords

ASLR; randomization; information leak; code reuse attack

CCS'15, October 12–16, 2015, Denver, Colorado, USA.
© 2015 ACM. ISBN 978-1-4503-3832-5/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2810103.2813694.

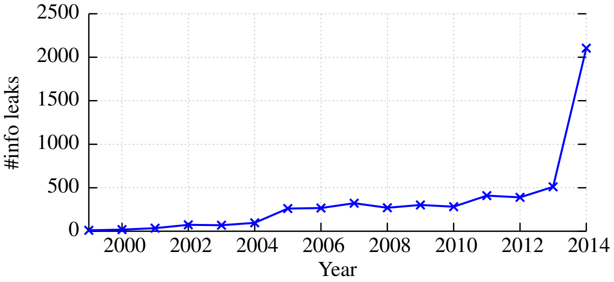

Figure 1: The number of information leak vulnerabilities reported by CVE Details[1]. There was a huge jump in 2014 to over 2,000 bugs that allow attackers to bypass ASLR.

1. INTRODUCTION

Since W⊕X became a de-facto security mechanism in modern operating systems, code injection is no longer a viable general attack technique. In particular, attackers can no longer place arbitrary shell code in the target process' *data* space and hijack control to start executing that shell code. Instead, attackers now rely on code reuse attack that hijacks the control flow to *existing code* in the target process (in an unintended manner). Ever since their first introduction, code reuse attacks have evolved from simply jumping to some sensitive library functions (a.k.a. return-to-libc) to chaining up small snippets of existing code (a.k.a. gadgets) with mainly returns and indirect calls/jumps to allow the attacker to perform arbitrary computations [38]. However, there are two prerequisites for these attacks: (1) a prior knowledge of the location of existing code gadgets, and (2) the ability to overwrite some control data (e.g. return addresses, function pointers) with the correct values so as to hijack the control flow to the targeted gadgets.

Address Space Layout Randomization (ASLR) aims to make the first prerequisite unsatisfiable by making the addresses of code gadgets unpredictable. In theory, ASLR can complement W⊕X and stop code reuse attacks effectively. By randomizing the location of different code modules, and even various instructions within code modules [22, 27, 32, 43], attackers without *a priori* knowledge of how the randomization is done will not know the location of the code that they want executed. As a result, attempts to launch code reuse attacks will likely direct the hijacked control to a wrong location and cause the attack to fail (usually with a program crash). It has been widely used in modern operating systems, and proven to be extremely efficient.

However, research in recent years has shown that many implementations of ASLR fail to live up to its promises of stopping code reuse

[1] http://www.cvedetails.com/vulnerabilities-by-types.php

attacks [7, 17, 37, 39, 41]. In some cases, the ASLR implementations fail to provide enough entropy thus are subject to brute-force attacks [7, 17, 39]. A more serious and fundamental problem for randomization based protection mechanisms is information leak vulnerabilities [37, 41], which breaks the implicit assumption that attackers cannot learn the randomness. With information leak, the attacker usually can obtain a "post-randomization" pointer to the location of a known module (e.g. specific function in a particular library) [7, 17, 39]. For ASLR that works at a module-level granularity (i.e., it does not modify the relative location of different instructions within a module), this kind of leak will allow an attacker to identify all target addresses within that module. Even for instruction-level randomization, with a sufficiently powerful information leak vulnerability, attackers can exploit a single code address leak and repeatedly read code pages to eventually locate all the code gadgets necessary to launch even the most sophisticated code reuse attacks [40]. More devastating, information leak is quite prevalent and the number of instances discovered is on the sharp rise. For example, there are over 2000 information leak vulnerabilities found in 2014 (Figure 1).

One way to solve this problem is to completely prevent code address leak. However, detecting or preventing code address leak is a very challenging problem. Due to the complexity of programs written in low level programming languages, it is even not clear about if one can find all pointers or data that can be leveraged to infer the code address. Moreover, code pointer and data can be inter-casted during the data propagation. And some code pointers (e.g., return addresses) are frequently dereferenced. Therefore, it is very challenging to correctly and efficiently protect code pointers.

In this paper, we propose ASLR-GUARD, a system that completely prevents code address leak to stop code reuse attack. The main idea is to either prevent code pointer leak or render any leaked code pointer useless in the construction of code reuse attack, so that we can reclaim the benefits of ASLR and prevent code reuse attacks. In particular, we propose three techniques: (1) completely decouple code and data by remapping all code to a random address, so a data pointer cannot be used to infer the location of code; (2) store all sensitive code locators (Hereafter, we use the term **code locator** to refer to *any pointer or data that can be used to infer code address*) in a secure storage, and (3) whenever a *code locator* is going to be propagated to the regular memory regions, we encrypt it to prevent attackers from exploiting the value of this code locator or using this code locator to hijack the control flow to arbitrary addresses.

We have implemented a prototype system ASLR-GUARD, which consists of two main components, a modified compilation toolchain that includes compiler, assembler and linker; and a modified dynamic loader. The first component is responsible for instrumenting an input program (in source code) and all its libraries so that all code locators can be identified and protected by our policies. The runtime component allows us to perform necessary initialization of our environment. Our evaluation results show that (1) ASLR-GUARD can thwart all the code locator leaks we have tested, even if attackers can dump all data regions; (2) with the capability of resisting against code locator leak attacks, ASLR-GUARD can reinforce ASLR to stop advanced code reuse attacks [7]; and (3) it incurs almost no runtime overhead (< 1%) on the SPEC benchmark.

In summary, the contributions of this work are:

1. We perform a systematic analysis to identify all sources of sensitive code locators, and implemented a memory checking tool to verify that we did identify and protect all code locators.

2. We propose a hybrid approach that employs isolation to protect most code locators (for better performance and security), and encrypt the remaining code locators that are hard to isolate.

3. We have implemented a prototype system, ASLR-GUARD, and our evaluation shows that ASLR-GUARD incurs a negligible runtime overhead on the SPEC benchmark and left no single code locator unprotected.

A point worth noting is that protecting pointer by encryption is a general approach, but the challenge is how to achieve both security and efficiency. For example, PointGuard [12] sacrifices security for performance by XORing all pointers using a single key. Such scheme is not secure against chosen-plaintext attacks. Further, PointGuard is also vulnerable to forgery attacks due to missing integrity checks. AG-RandMap overcomes these problems by proposing a novel and efficient encryption scheme (Figure 3). Furthermore, as we will show in §3, our work overcomes a major technical challenge of identifying all the code locators that need to be encrypted. Finally, our policy for handling code locators on stack allows us to improve both the security and, very significantly, the performance of ASLR-GUARD.

In the rest of the paper, we will introduce our threat model (§2), present the results of our effort to identify all code locators to be protect (§3), describe the design and implementation of ASLR-GUARD (§4, §5), evaluate our approach (§6), discuss the limitations of, and future work for ASLR-GUARD (§7), compare ASLR-GUARD with related work (§8) and conclude (§9).

2. THREAT MODEL

To make sure our solution is practical, we define our threat model based on strong attack assumptions, which are commonly used in projects related to code reuse attacks [7, 17, 39, 40]. As the trusted computing base (TCB), we assume a commodity operating system (e.g., Linux) with standard defense mechanisms, such as non-executable stack and heap, and ASLR. We assume attackers are remote, so they do not have physical access to the target system, nor do they have prior control over other programs before a successful exploit. We assume the platform uses 64-bit virtual address space, as 32-bit address space cannot provide enough entropy and 64-bit processors are widely available.

For the target program, we assume it is distributed through an open channel (e.g., Internet), so attackers can have the same copy as we do. This allows them to do any analysis on the program, such as finding vulnerabilities and recomputing all possible code gadgets. We assume the target program has one or more vulnerabilities, including control-flow hijacking or information leak vulnerability, or both kinds at the same time. More specifically, we assume the program contains at least one vulnerability that can be exploited to grant attackers the capability to read from and write to an arbitrary memory address. We further assume this vulnerability can be exploited repeatedly without crashing the target program, so attackers can use the *arbitrary memory read/write* capability at will. Since arbitrary memory read/write on the 64-bit virtual address space is probabilistically impossible to achieve attacker's goals, we assume all arbitrary memory read/write will be based off on the memory addresses leaked from the previous vulnerability (or second-order guess based on the leaked code locator). Finally, we assume the ultimate goal of the attackers is to divert the control flow and execute arbitrary logic of their choice.

Although there are a few known explicit leak channels (e.g., `/proc/self/maps`), many security enhanced Linux distributions such as PaX disable `/proc`-based pointer or layout leaks. We assume there is no explicit leak through the platform itself.

Out-of-scope threats. Since we assume the OS as our TCB, we do not consider any attack that tries to exploit an OS vulnerability to gain control over the target program. Given our threat model, we focus on preventing the attacks that exploit vulnerabilities to bypass ASLR, and then overwrite control data to hijack control-flow. Non-control-data attacks [10] (e.g., hijacking credential data or metadata of code locators, like object pointers) are out of our scope. We also do not consider information exfiltration attacks.

3. CODE LOCATOR DEMYSTIFIED

Since many kinds of data besides code pointer can be leveraged to infer the address of code, such as the base address of `text` section and offset in a jump table, we use the term *code locator* to refer to any pointer or data that can be leveraged to infer code address. Although many previous works like the JIT-ROP attack [40] have shown that a single code locator leak may compromise the whole ASLR protection, to the best of our knowledge, there was no existing work that systematically discussed the composition of code locators. In order to provide a comprehensive protection against information leak, we first conduct a systematic analysis to discover all code locators in a program, and discuss the completeness of coverage.

3.1 Discovery Methodology

When a program is starting, the kernel is responsible for creating the address space of the process and loading the ELF binary and dynamic linker, while the relocation of code pointers is left for dynamic linker. Dynamic linker then loads all required libraries and performs necessary relocations before the loaded code is executed. Once all relocations are done, the dynamic linker transfers the control to the program's entry point. During execution, the program may (indirectly) call function pointers, or interact with OS, e.g., signal handling. We categorize code locators mainly based on the life cycle of the program execution. Then we try to identify all code locators at different program execution stages. More specifically, we first thoroughly checked the source code of dynamic linker (`ld.so`), libc libraries, and gcc to understand how code locators could be generated at load-time and runtime of the program execution. As for code locators that are injected by the OS into the process' address space, we implemented a memory analysis tool to exhaustively check if *any 8-byte* in readable memory points to any executable memory in the target program or bases of modules, and we performed this check before and after executing every system call. The memory analysis tool also serves to validate our process for discovering code locators injected by the dynamic linker and the program itself (e.g., checking the memory at the entry/exit points). Once our memory analysis tool discovers new kinds of code locators, we manually verify and categorize them. We then append the new "category" to the identified code locator set, and run memory analysis tool again to find more kinds of code locators. The iterative process ends until no new code locators are reported. We summarize our results of such exhaustive memory analysis in Table 1.

3.2 Code Locators at Different Stages

3.2.1 Load-time Code Locators

During load-time, there could be various kinds of code locators computed and stored in the memory, e.g., static function pointer and `.gotplt` entry, as shown in §3. Although it is hard to iterate all code locators produced at load-time, we found the fact that, suppose ASLR is fully enabled, all code locators must be relocated before being dereferenced. With this fact in mind, we designed a general approach to discover and protect `all` load-time code locators. We control the relocation functions in dynamic linker, so that all load-time code locators must go through our relocation, and therefore we can enforce our protections for them, e.g., isolation and encryption.

3.2.2 Runtime Code Locators

Code locators can be generated at runtime by deriving from the program counter (e.g., `RIP`). In x86_64 platform, the `RIP` value can be loaded either by a `call` instruction or a `lea` instruction. `call` instruction pushes return address on the top of stack, and when the callee returns, the return address is dereferenced. Usually, people believe `call` and `ret` are paired. In our analysis, we indeed found some corner cases in which return address is not used by `ret`. As the code locator type **R3** shown in §3, `setjmp` loads return address using a `mov` instruction, and the return address is then dereferenced as a function pointer by `longjmp`. Besides the return address, runtime code locators can also be generated by `GetPC` (e.g., `lea _offset(%rip), reg`). For example, if there is a function pointer that is assigned with an address of a local function, typically `GetPC` will be used to get the function address. Since we have the complete control of the compilation toolchain, we could properly discover `all` runtime code locators (e.g., the ones introduced by `call` or `GetPC`) and enforce the corresponding protections, which are elaborated in §4.

3.2.3 OS-injected and Correlated Locators

As ASLR-GUARD is designed to work with unmodified commodity kernel, how the program interacts with kernel needs be analyzed in order to discover code locators that might be injected or transmitted by the kernel. We used our memory analysis tool to check the memory right before and after a syscall. Based on our analysis, we found that the program entry pointer can be stored on the stack by kernel. Also kernel may save the execution environment, including `RIP` value, for signal/exception handling. Since data sections and code sections are mapped together, the offsets between them are fixed. Data pointers can be leveraged to infer code addresses. We call such pointers the correlated code locators, which we correctly handle to prevent intentional code address leaks.

3.3 Completeness of the Analysis

In an ASLR-enabled program, all static code locators (e.g., the ones in `.gotplt` and virtual table) must be relocated at load-time. As we can control the relocation routine, we can guarantee to cover all such code locators. Another source of code locator at load-time is that a code locator is calculated based on a base address. So we also collect all cases in dynamic linker that access the base address stored in `link_map`. In this way, we can cover all load-time code locators. Note that we will remap all code sections to random addresses, any missing code locator at load-time (although we did not meet such case in our evaluation) will crash the program, which means our analysis for load-time code locators is "fail-safe". For runtime code locators, as we have the control of whole toolchain, we can easily guarantee to find all runtime code locators (i.e., we catch all instructions that access return address or `rip`).

As for code locators that are injected by the OS, we take a dynamic analysis approach. While this dynamic analysis approach is not necessarily complete in theory, we will argue that in practice the way the OS setups a process' address space (and thus all the bookkeeping code locators it may inject into various user space data structure) should be the same for every process; a similar argument should apply for any code locator injected for signal handling.

4. DESIGN

In this section, we present the design of ASLR-GUARD. Based on previous discussion and existing attacks [7, 35, 40], we know

Category	Type	Description	Applied protection
Load-time locators	**L1**. Base address	Base addresses of loaded modules by OS	Encryption + Isolation
	L2. GOTPLT entry	Library function pointers or address of `_dl_runtime_resolve`	Isolation
	L3. Static/global pointer	Contained in some data sections (e.g., `.init_array`)	Encryption + Isolation
	L4. Virtual function pointer	Contained in C++ virtual tables	Encryption + Isolation
Runtime locators	**R1**. Return address	Pushed on stack by call instructions	Isolation
	R2. GetPC	A program counter address is loaded	Encryption
	R3. GetRet	A return address is loaded as a function pointer	Encryption
OS-injected locators	**O1**. Entry point	Address of entry point (e.g., `_dl_start_user`)	Encryption
	O2. Signal/Exception handler	All registers including program counter are saved in memory	Encryption
Correlated locators	**C1**. Jump table entry	Offsets from jump table to code blocks	Isolation
	C2. Data pointer	Any pointer pointing to data with fixed offset into code section	Decoupling data/code sections

Table 1: A summary of all code locators that can be used to infer code address. All of them are protected by ASLR-Guard

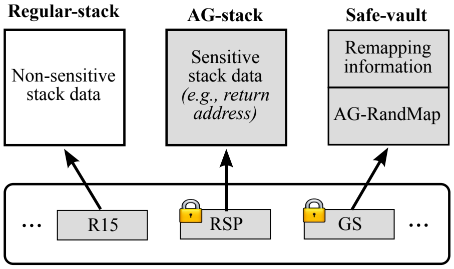

Figure 2: Register usage semantics in ASLR-Guard. The usage of `rsp` and `gs` will be securely regulated by ASLR-Guard's compiler, so never be copied to the program's memory, nor accessed by the program's code.

that as long as attackers can exploit information leak vulnerabilities to obtain a valid code locator, ASLR will continue to be nullified, no matter how much we improve the granularity of the randomization. So the goal of ASLR-Guard is to: *completely prevent leaks of code locators*.

Figure 2 demonstrates the high-level idea of ASLR-Guard. First, it decouples code sections and data sections. By doing so, the process' address space is virtually divided into two worlds: the data world and the code world. This step eliminates all the implicit relationship between data and code (C1). Second, ASLR-Guard acts as the gateway between these two worlds: whenever a code address is about to be leaked to the data world, ASLR-Guard encrypts the address as a token; and whenever a token is used to perform *control transfer*, ASLR-Guard decrypts it. By doing so, ASLR-Guard eliminates all explicit code locator leaks.

To implement this scheme, we developed four key techniques: *safe vault*, *section-level randomization*, *AG-RandMap* and *AG-Stack*.

4.1 Safe Vault

Safe vault is an "isolated" memory region where all the plaintext code locators (except `R1`) listed in Table 1 are stored. We divide the safe vault into two parts. The first part is for storing the information of all remapped sections, which includes the base address for each section before and after the remapping, as well as the size of each section. The second part is the random mapping table used for code locator encryption/decryption.

In ASLR-Guard, we guarantee the isolation of the safe vault with the following design. First, the base address of the safe vault is randomized via ASLR. Second, the based address is stored in a dedicated register that is never used by the program [2], and its content is never saved to the data memory. As a result, one can only locate the safe vault by brute-forcingly guessing its base address, which is prohibited by ASLR on 64-bit systems.

4.2 Section-level randomization

The purpose of the section-level randomization is two-folded: first, we use it to stop attackers from deducing the location of our code using leaked data pointer; second, we would like to further protect data structures that contain code locators (namely jump tables, `.gotplt` tables and virtual function tables) without paying the price of encrypting all the code locators in them.

For our first goal, we decouple code from data by remapping all code sections (code sections can be identified by looking up ELF header) to random addresses. Since the offset between code and data is changed during this process, we further adjust the constant offsets in all data-access instructions. This is done at load time based on the relocation information generated by our linker. For our second goal, we modified the compiler to make sure that: (1) jump tables are always emitted in `rodata` sections, (2) addresses pointing to, or derived from the content of, code locator tables like jump tables and `.gotplt` tables always stay in registers and are never saved to the memory. With these two guarantees, we can protect code locators in these data structures by simply remapping them to random addresses, without ever needing to encrypt the code locators stored in these tables. One exception to our second rule is the virtual function tables. Since we cannot guarantee that virtual function pointers will never be stored to the unsafe data world, we encrypt virtual function pointers after the virtual tables are remapped to a random address. After the remapping, we use the safe vault to store the remapping information for each of the protected section so we can later patch up accesses to them. One final point to note is that to make sure the remapped location of the various tables are truly random, we wrapped the *mmap()* function to provide it random base addresses. As the primary goal of ASLR-Guard is to harden ASLR, instead of to improve the entropy of ASLR, we set the same entropy as in the default 64-bit Linux ASLR (28-bit). We acquire the randomness from cryptographic secure source, i.e., the `RdRand` instruction on modern Intel processors. There are alternatives from `/dev/random` that could be used.

4.3 AG-RandMap–Locator Encryption

In this subsection, we define our encryption scheme for encrypting code locators. This encryption scheme is designed to achieve two goals. First, it must be strong enough to stop code reuse attacks even when some (or all) of the encrypted code locators are leaked. We also assume that attackers can know what functions the leaked code locators point to. In order to achieve this goal, our scheme

[2] On x84-64 platform, the segmentation register `gs` is the most suitable candidate for this purpose.

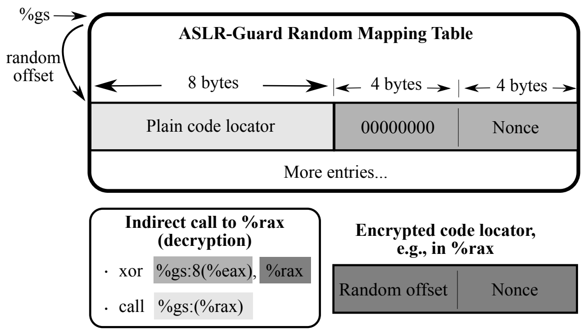

Figure 3: Code locator encryption/decryption scheme.

need to prevent attackers from efficiently deriving valid encrypted code locators that will be dereferenced to direct the control flow to code gadgets. In particular, our encryption scheme should make the task of deriving a valid encrypted code locator as difficult as brute-forcingly guessing the gadgets under ASLR. The second requirement for our encryption scheme is performance, especially for the decryption operation. This is because code locator generation is a relatively rare operation, but code locator dereferencing (e.g., indirect call) is much more frequent.

To achieve these goals, we propose AG-RandMap, an efficient `random mapping table` mechanism, as our encryption scheme. The main idea of AG-RandMap is illustrated in Figure 3. To encrypt a code locator, we will randomly pick an unused 16-byte entry in the mapping table. The first 8 bytes will be used to store the plain code locator being encrypted; the next 4-bytes is zero that is used to get the random offset with XORing during decryption, and the last 4-bytes contains a random nonce (with 32-bit entropy) for integrity check. As in 4.2, we acquire the randomness from cryptographic secure source (i.e., the `RdRand` instruction provided by modern Intel processors). After building the new table entry, the 4-byte random offset from the base of the mapping table to the new entry is concatenated with the nonce, and returned as the encrypted code locator. Note that AG-RandMap does not use any key for encryption, but generates an entry in the mapping table and saves the encrypted code locator in the corresponding register or memory using "one-time" random value.

Decryption under our scheme is performed very efficiently and involves only one extra `xor` operation on top of the original indirect control transfer. The code locator decryption process is also illustrated in Figure 3. Assuming the encrypted code locator is in `%rax`, we first use `%eax`-plus-8 as an offset to read 4-bytes zero and the nonce from the table, and `xor` it with the encrypted code locator. If the nonce is correct, the result will be a valid offset within the mapping table, which then decrypts `%gs:(%rax)` to the plain code locator. Otherwise, the offset will be outside of the table, transferring the control to a random address that is highly likely to be either unmapped or not executable, thus crashing the program. In other words, for an attacker to forge a valid encrypted code locator that will direct the control flow to a plaintext address stored in our AG-RandMap, he will need to know the correct nonce to use for the target table entry. To do so, he can either guess with a successful chance of 2^{-32}; or try to locate our AG-RandMap, which is equivalent to brute-forcing ASLR. Either way, our encryption scheme satisfies the first requirement.

C++ support. To support polymorphism, C++ uses a virtual function table (vtable) to store the pointers of virtual functions for each class with virtual functions. And in each object of class with virtual function, vtable pointer(s) (vptr) are used to point to the right vtable. ASLR-Guard generally protects the function pointers inside vtable by encryption. However, as vtable pointers are stored in objects in unsafe memory, leaking vptr may help attackers know all encrypted virtual function pointers and allow COOP attack [35]. To prevent such attack, ASLR-Guard further encrypts vtable pointers so that attack cannot even know the locations of encrypted virtual function pointers, which is an essential step for COOP attack[35].

4.4 AG-Stack

As discussed in §3, some dynamic code locators are stored on the stack, such as return addresses (R1) and OS injected code locators (O2). To prevent leaking return address through stack, one approach is to apply the same encryption/decryption scheme in §4.3. However, this approach is not suitable for protecting return address on the stack. First of all, the return addresses are generated very frequently, and our encryption scheme does not support frequent encryption very well. Second, it will require extensive modification to support encryption over OS injected code locators. For these reasons, we propose an efficient alternative solution, namely AG-Stack to protect R1 and O2.

Our approach is similar to the traditional shadow-stack techniques [18, 34] that are used to prevent return addresses from being modified by attacks. However, they protect return addresses by costly instrumenting `call/ret` pairs. For example a single `ret` is instrumented with at least 4 instructions that load return address in the shadow stack, adjust the stack, check equivalence, and `ret`. Therefore, traditional shadow-stack approaches face three problems: (1) `call/ret` are frequently executed by CPU, instrumenting them incurs significant performance overhead; (2) cases where `call/ret` don't come in matching pairs, e.g., `GetPC`, are difficult to handle; (3) "return addresses" injected by the OS will not be automatically protected. To address these issues with traditional shadow-stack techniques, we propose a novel mechanism, namely AG-Stack.

Besides being much more efficient, AG-Stack has several advantages: (1) it provides the capability to store more sensitive data; and (2) it does not change the semantics of original code, so special cases like unpaired `call/ret` instructions and signal handling can be naturally supported. The high level idea of AG-Stack is to maintain two stacks by re-arranging the usages of two registers. Specifically, we use the ordinary stack (i.e., the one accessed through `RSP`, as shown in Figure 2) as our AG-Stack to store sensitive on-stack data like return addresses and function pointers used for exception handling. All other data, like stack variables, register spilling and function call arguments are stored on a regular stack that accessed through `R15`, as shown in Figure 2. The security of AG-Stack is guaranteed in a way similar to safe vault. As in safe vault, its base address is randomized. Also, since AG-Stack does not store any program data, there is no data pointer pointing to it, so attackers cannot derive its address through memory traversal. Finally, whenever the stack pointer is stored in memory (e.g., during `setjmp`), we will encrypt it using the scheme in §4.3. As such, with AG-Stack, we achieve a better performance but a guaranteed security for all code locators saved on the stack.

AG-Stack is very similar to the safe-stack from CPI [26]. Both impose no runtime overhead. The difference is that safe-stack performs type-based analysis and intra-procedure analysis to find unsafe stack objects. As shown in Section §3, type-based analysis may not cover some non-pointer code locators. AG-Stack eliminates these analyses by thoroughly re-arranging the register usages.

Multi-thread support. In Linux, multi-thread feature is enabled with Native POSIX Thread Library (NPTL), which creates a new thread with the `clone` system call. Specifically, to create new thread, NPTL first allocates a new stack and stores thread descriptor at the

Component	Tool	Lines of code
Compiler	gcc-4.8.2	120 lines of C
Assembler	as-2.24	900 lines of C
Linker	ld-2.24	180 lines of C
Dynamic Linker	eglibc-2.19	1,200 lines of C
Memory analyzer		800 lines of Python
	Total	3,200 lines of code

Figure 5: Toolchain modifications made for ASLR-Guard.

end of the new stack. And then it saves the thread function pointer and its arguments in the new stack. After that, `clone` system call switches the context, and the thread function is called. Similarly, to support multi-thread in ASLR-Guard, for each new thread, we allocate a regular stack (with the same size) whenever the new (safe) stack is allocated, and release the regular stack when the thread exits. The regular stack information (e.g., stack address and size) is saved in thread descriptor that is stored in AG-Stack.

5. ASLR-Guard TOOLCHAIN

In this section, we describe our prototype implementation of our protection scheme. We first give an overview of our prototype, then we discuss the detail of each component of ASLR-Guard.

5.1 Architecture

Figure 4 shows the overview of ASLR-Guard's architecture. To remove all code locator leaks, ASLR-Guard requires instrumenting all loaded modules, including modules of the target program as well as all shared libraries. In this paper, we assume the access to the source code of the target program is available and the instrumentations are done through re-compilation.

ASLR-Guard consists of two major components: the *static toolchain* and the *runtime*. The static toolchain is in charge of (1) implementing the AG-Stack, which will be used to securely store return address and OS injected code locators; and (2) instrumenting the program to correctly encrypt runtime code locators and decrypt all code locators. The ASLR-Guard runtime contains three majors parts. The first part is the dynamic loader, which is in charge of (1) initializing the AG-Stack and the safe vault, (2) decoupling code sections from data sections, and (3) encrypting all load-time code locators. The second part is the standard `C/C++` libraries that will be linked into most programs. Among these libraries, some need special care because they need to handle some OS-injected code locators in special ways (e.g., `setjmp`).

We implemented our prototype based on the GNU toolchain: `gcc`, `binutils` and `glibc`. Figure 5 summarizes the our efforts of modifying the GNU toolchain.

5.2 Compiler

We modified the GNU compiler `gcc` to assist implementing the code locators encryption and AG-Stack. For code locator encryption, we let `gcc` insert a flag for data-read instructions of global data pointers to differentiate global function pointers in the generated assembly code. For AG-Stack part, we performed several modifications in `gcc`. First, we need to reserve a register for the unsafe stack. In `gcc`, this can be done through the `-ffixed` option. But a more important question is, which register should be chosen. In ASLR-Guard, we choose this register using two criteria: (1) it should be one of the least used general registers in handwritten assembly code, and (2) it should be one of the callee-saved registers (i.e., non-volatile). The first requirement is for that compiler can only guarantee that the code generated by them does not use the reserved register, however, any handwritten assembly (including inline assembly) may still use this register. They need to be modified to use other registers (e.g., we found and modified about 20 cases in `glibc`). So choosing the less frequently used register helps us minimize the modifications. The second requirement is for compatibility with legacy uninstrumented binary. Because we reserved the `RSP` for AG-Stack, by using a callee-save register, it ensures that both code can work correctly. Based on these criteria, we chose `R15` as the register for the regular stack on our target platform.

Second, we want to make sure that besides `call/ret`, there is no implicit `RSP` modifications. On x86-64, such modifications can be introduced by instructions, `push/pop`, `enter/leave` and `call/ret` [23, Ch. 6.2]. `Push/pop` are for saving/restoring the target registers, and `enter/leave` is a shortcut for saving/restoring the frame register. Our elimination of `push/pop` is done in two steps. We first leveraged existing compiler optimizations to reduce the usage of `push/pop`. In particular, modern compilers like `gcc` in most scenarios prefer using `mov` instruction for performance gain, as CPU can do pipeline scheduling for `mov` instructions. So we modified `gcc` to always prefer using `mov` instruction for passing arguments (as if `-maccumulate-outgoing-args` and `-mno-push-args` are always set) and saving/restoring registers at function prologue/epilogue. For the remaining `push/pop` operations, our assembler will replace them with corresponding `mov` operations on `R15`. The elimination of `enter/leave` is done similarly

Finally, we need to re-align the stack. The System V AMD64 ABI uses `XMM` registers for passing floating point arguments. And reading/writing these registers from/to memory requires the memory address to be 16-byte aligned. As we split the stack into AG-Stack and regular stack, the regular stack needs to be re-aligned. We enforced the alignment by leveraging `gcc`'s supports for multiple platforms. Specifically, on platforms like ARM, function invocations do not automatically save the return address on stack, so `gcc` already understands how to align stack when there is no return address on stack. Using this support, we modified `gcc` to treat our target platform as if it does not explicitly save the return address on stack (by setting `INCOMING_FRAME_SP_OFFSET = 0`).

5.3 Assembler

We modified the GNU assembler `gas` to perform code instrumentations for code locator encryption/decryption and AG-Stack. The first task is to encrypt every runtime code locator (type R2, R3 in Table 1). For position-independent ELF executables, runtime code locators are created through retrieving the `PC` value (e.g., `mov (%rsp), %rax` may load return address) or a PC-relative address (e.g., `lea offset(%rip), %rax`). ASLR-Guard uses these signatures to identify all potential code locators. Specifically, we perform a simple intra-procedure analysis (check if `RSP` points to return address) to find all instructions that load return addresses. And we find the instructions of loading "PC-relative" code locators based on the metadata (e.g., indicating if a symbol is a function) contained in the assembly files and the global data pointer flag inserted by our compiler 5.2. Note that, for handwritten assembly, there is no easy way to distinguish the ones for global data access and global code access at assemble time. To address this issue, our assembler first conservatively instruments both of them, and defer it to our linker to remove the false positive ones, as linker has code boundary information to verify code locators. For each identified instruction, our modified assembler will insert an encryption routine to immediately encrypt the code locator, as shown in Figure 3

Next, we instrument the target program to correctly decrypt code locators. Specifically, we first instrument every indirect call with code address in register in the ways shown in Table 3. If the target

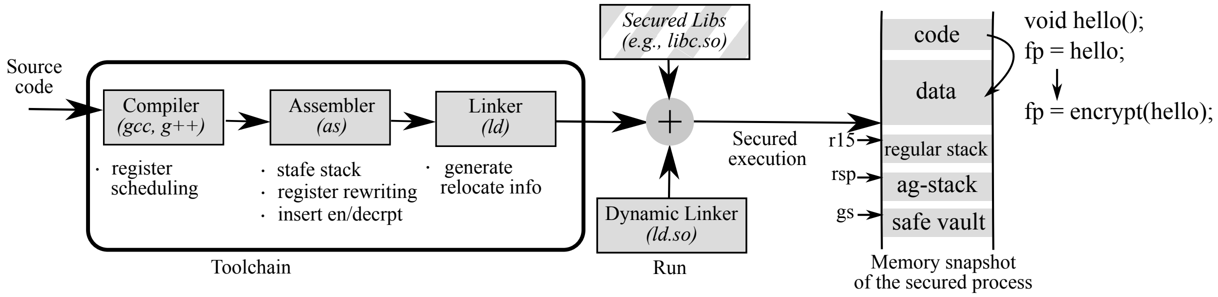

Figure 4: An overview ASLR-GUARD's architecture. It consists of two components: toolchain (e.g., compiler) and runtime (e.g., loader and libc). Code locators will never be copied into the program's data memory (see right). Therefore, the secured program will be free from memory leakage vulnerabilities that might directly or indirectly break the applied ASLR hardening.

is stored in memory, we first save `RAX` and load the target into `RAX`. Then we perform the decryption as the one in Table 3. One difference is that, to recover `RAX`, we temporarily store the decrypted code address in AG-Stack, do recovery, and call into the code pointer in AG-Stack. Indirect jump for invoking function is instrumented in the same way.

The third task is to replace all `push/pop` and `enter/leave` operations with explicit stack pointer operations. After this, ASLR-GUARD will replace all `RSP` occurrences with `R15`, except when `RSP` is used to access return address in handwritten assembly. Using the same intra-procedure analysis mentioned above, we can find all such cases and skip the replacements.

5.4 Static Linker

We modified the GNU static linker `ld` to perform two tasks. The first task is to provide relocation information to the dynamic linker. This is necessary because once we decoupled code sections from data sections, existing data access instructions will no longer work because the offsets have changed. To fix this problem, ASLR-GUARD creates a new relocation table similar to the `.rela.dyn` to provide this information, i.e., which 4-bytes need to be patched. The second task is to remove encryptions for global data access in handwritten assembly, conservatively added by assembler. This task is performed here because the linker has the information of all static modules, so it can identify whether an instruction is for accessing data or code.

5.5 Dynamic Linker

We modified the dynamic linker `ld.so` of `glibc` to initialize the runtime, decouple code sections from data sections, and encrypt load-time code locators. When running an executable, Linux will first load its interpreter (i.e., the dynamic linker), jump to the entry of its entry and let the interpreter handle the target binary. This means the dynamic linker is the very first module to be executed. For this reason, we insert our runtime initialization code at the entry of the dynamic linker (`_dl_start()`). The regular stack is initialized by allocating a new memory region with a random base, and copying all the arguments and environment variables from the initial stack. Safe vault is setup by allocating another random memory region and assigning the `gs` segment register to its base address. Once all the initializations are done, the control is transferred to the remapped code of dynamic linker, and the original one is unmapped.

The decoupling is done by remapping the corresponding sections to random addresses as the way mentioned in Section 4.2. After remapping, ASLR-GUARD will encrypt all load-time code locators (see Table 1). This is done during the relocation patching. Specifically, after randomizing the base address, the original dynamic linker already provides some relocation functions to patch all those load-time code locators with correct code addresses. In ASLR-GUARD, with the help of this feature, we modify these relocation functions to patch the load-time code locators with encrypted values, instead of patching them with the real code addresses.

5.6 Standard C/C++ Library

Protecting the standard C/C++ libraries is very important because they are linked into every `C/C++` program. Yet they contain handwritten assembly that may need special care, to handle some corner cases may have compatibility problems with the automated hardening techniques. Unfortunately, many previous work did not show enough discussions on handling these libraries. In this subsection, we try to shed some light on this topic.

We first compile `eglibc-2.19` and `gcc-4.8.2` with our own toolchain. Then we handle three main cases in handwritten assembly code: (1) access to return address and stack parameter; (2) stack layout mismatching; (3) indirect jump for invoking functions; In the first case, we use our intra-procedure program analysis to find instructions for loading return address (e.g., `mov (%rsp), %rax`). For each identified instruction, we insert the encryption routine right after it. However, because the regular stack does not contain the return address any longer, the offsets used to access stack parameters (x86_64 calling convention passes parameters via stack after the first 6 ones) are incorrect. We address this by manually adjusting the regular stack top at function prologue and epilogue. In total, we found 12 such cases in the handwritten assembly of `glibc`. The second case happens when the caller is in handwritten assembly and callee is in C code, or vice versa. For example, `_dl_runtime_resolve` is written in assembly, at the end of which, its local variables, return address, and parameters are released at once by a stack-pivot instruction (i.e., `addq $72, %rsp`), as it assumes there is only a single stack. However, this is not correct as AG-Stack and regular stack are supposed to be released separately. To handle this, we manually change the assembly. In total, we handled 4 such cases in libc. As mentioned in §4.2, indirect jumps for jump table will not be instrumented, but the ones for invoking functions will be. To differentiate them in handwritten assembly, we manually verify if a jump instruction is used for function invocation or not, if yes, we instrument them to go through the code locator decryption. We found 1 case in `glibc`.

`DWARF` standard is adopted for C++ exception handler (EH). For each `throw` statement, an exception is allocated and the stack unwinding (i.e.,`unwinder`) is started. For each `catch` statement, `g++` has already generated a EH table and a cleanup table during compilation. And at runtime, the `unwinder` will walk through the stack twice to handle the exception: (1) trying to find the correct exception handler based on EH table; (2) if a matching is found, walking though the stack a second time for cleanup based on the cleanup table. To support C++ exception handling in ASLR-GUARD, we need to update these two tables and rewrite the `unwinder` to support AG-Stack. Since doing so merely requires some engineering efforts and many C++ programs do not use it (e.g., Chromium does not use it and only two SPEC benchmark programs use it), we did not support C++ exception handling in our current prototype yet, but leave it for future work.

6. EVALUATION

In this section, we present the evaluation of ASLR-GUARD. Our evaluation tries to answer the following questions:

(1) How secure is the ASLR-GUARD's approach in theory, compared to general ASLR approach? (§6.1);

(2) How secure is the ASLR-GUARD's approach against practical memory-based attacks, empirically? (§6.2);

(3) How much overhead does ASLR-GUARD impose, in particular runtime, loading time, and space? (§6.3)

Experimental setup. We carried out evaluations of ASLR-GUARD to check its security enhancements and potential performance impacts. Our evaluations are mainly performed on the standard SPEC CPU2006 Benchmark suite. Beside that, we also applied ASLR-GUARD to complex and real world programs, e.g., `gcc`, `glibc` and Nginx web server. We compiled all above programs and their libraries with the toolchain of ASLR-GUARD and enforced them to use the dynamic linker of ASLR-GUARD. All programs were run on an Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz machine with 128 GB RAM.

6.1 Security Analysis

In this subsection, we will present our analysis to determine the probability for an attacker to hijack the control to a particular target address (let's say, `x`). For our discussion in this section, we will assume the ASLR provided by the underlying OS (in our implementation, 64-bit Linux) provides 28-bit entropy for the address of any section that has been randomized.

Referring to Table 1, we can see that if all code locators to `x` are generated from sources **L2**, **R1** or **C1** (i.e. items only reachable through `.gotplt` tables, return instructions or jump tables), the attacker can only hijack the control to `x` by overwriting the content of the safe vault. This is because ASLR-GUARD will never generate any encrypted code locator for `x`, thus attackers cannot achieve the goal by overwriting an encrypted code locator. As such, the probability for successful hijacking is the same as that of breaking ASLR, i.e. 2^{-28}, assuming the attacker knows the semantics, e.g., the offset of `x` in memory page.

On the other hand, if `x` can be reached through an encrypted code locator (i.e. `x` is the address of items in the categories **L1**, **L3**, **L4**, **R2**, **R3**, **O1**, **O2** of Table 1), and assuming an encrypted code locator for `x` exists in AG-RandMap[3], attackers have a new option by trying to overwrite some encrypted code locators residing in the unsafe memory with a value that can "decrypt" the control to `x`. For this kind of target, given a leaked encrypted code locator other than `x`, as we have argued in §4.3, and the attacker cannot read the content of AG-RandMap in safe vault, his chance of creating the correct encrypted value of `x` is 2^{-32}, guaranteed by the random nonce. Either way, the best chance of a successful attack is no more than 2^{-28}. We discuss a case of that the encrypted value of `x` is leaked in section 7.

6.2 Practical Security

To see how effective ASLR-GUARD is in practice, we performed multiple empirical security analyses.

Memory snapshot analysis. As mentioned in §3, even it is unlikely, we cannot guarantee kernel will never save code locators in unsafe memory, as our current design did not instrument kernel code. To make sure there is no single plain code locator left in the unsafe memory by kernel, we did a thorough memory analysis using our memory analysis tool 3. More specifically, we first applied ASLR-GUARD to SPEC benchmark programs. And then we hooked the entry/exit points of the programs, as well as all points right after syscalls, and dumped the whole memory core for memory snapshot analysis. The analysis results are shown in Table 2. As expected, there is no single plain code locator left in unsafe memory. All code locators are either isolated or encrypted, which prevents attackers knowing the address of any executable memory. Note that, our memory analysis is conservative and may contain false positives that some random data is falsely treated as code locator. Indeed, we found two "code locators" that point to executable memory, in `sphinx3`. However both code locators are eliminated due to they actually point to the the middle of instructions, so we believe they are false positives. Furthermore, we evaluated other numbers, e.g., the map table size, number of all encrypted code locators, number of load-time code locators, size of `.gotplt` tables, etc. Out of them, an interesting number is the one of encrypted code locators left in the unsafe memory. To get this number, we check if any 8-bytes is "pointing" to the random mapping table by decrypting it in the way shown in Table 3. Note that such analysis is conservative, and the actual number could be even less. As we can see in Table 2, in most programs, less than 10% encrypted code locators are propagated to unsafe memory. Many of them have less than 20 ones in unsafe memory.

Nginx web server. We chose nginx to evaluate the effectiveness of ASLR-GUARD, not only because it is one of the most popular web servers but also that it has been compromised by couple of attacks [7, 17]. We took Blind Return Oriented Programming (BROP) [4] as an example to show how ASLR-GUARD defeats the return address leak, which is an essential step in BROP attack. BROP attacks `nginx 1.4.0` (64-bit) by exploiting a simple stack buffer overflow vulnerability. This attack consists of three steps: (1) guessing the return address on the stack; (2) finding enough gadgets; and (3) building exploit. In step (1), BROP proposed using stack reading approach to overwrite the return address byte-by-byte with possible guess value, until the correct one is found. Naturally, ASLR-GUARD can prevent such attack at the first step, as return addresses are not stored in regular stack at all. To verify it, we applied ASLR-GUARD to nginx, and run the exploit again. As expected, the exploit failed at guessing the return address. Moreover, we did the memory analysis during this attack. We found there were 1,474 code locators encrypted, out of which, 361 ones were propagated to unsafe memory. Again, no plain code locator was left in unsafe memory.

Average indirect targets. If we conservatively assume the attackers can read the whole unsafe memory and understand the semantics of the memory (as will be discussed in §7), attackers may reused the leaked encrypted code locators to divert control flow. Given this conservative assumption, we want to know how many encrypted code locators are left in unsafe memory. Also we define a metric `Average Indirect Targets` (AIT) to measure the average number of possible targets an indirect branch may have. The reason we do not reuse AIR (Average Indirect target Reduction) metric proposed

[3] Otherwise, the situation will be the same as above, the attacker will have to overwrite values in the safe vault.

[4] http://www.scs.stanford.edu/brop/

Programs	Size of rand map table(byte)	# all enc code locators	# load-time code locators	# PLT entries	# fixed up PLT entries	# enc CL in unsafe mem	# plain CL in unsafe mem
perlbench	14,144	884	967	131	34	603	0
bzip2	4,480	280	303	28	11	20	0
gcc	22,784	1,424	1,339	89	39	187	0
mcf	4,528	283	361	39	18	9	0
gobmk	32,720	2,045	2,104	66	26	1,771	0
hmmer	4,544	284	361	83	37	10	0
sjeng	4,592	287	365	42	21	27	0
libquantum	4,512	282	324	46	12	9	0
h264ref	4,736	296	341	64	37	16	0
astar	14,560	910	2,289	1,348	33	29	0
xalancbmk	141,696	8,856	10,275	1,449	184	258	0
milc	4,544	284	336	58	15	9	0
namd	14,880	930	2,262	1,353	36	44	0
dealII	54,784	3,424	4,884	1,473	310	81	0
soplex	21,472	1,342	2,717	1,377	254	89	0
lbm	4,528	283	316	38	13	9	0
sphinx3	4,544	284	354	76	39	10	0
Average	**21,062**	**1,316**	**1,759**	**456**	**66**	**187**	**0**

Table 2: Code locator numbers. (1) Size of random map table, number of all encrypted code locators, number of fixed up `.gotplt` entries are collected when the programs exit, as they are accumulated. (2) Number of all static code locators and number of GOTPLT entries are collected at the point of main() is called. (3) Number of encrypted code locators in unsafe memory and number of plain code locators in unsafe memory are the maximum numbers among the ones collected after each syscall and at the points of main() and exit() are called. If there are multiple commands in `speccmds.cmd`, we calculate the average numbers.

Programs	Original-CFI		bin-CFI/CCFIR		AG	
	ret	call	ret	call	ret	call
perlbench	1	7,722	28,066	7,722	1	603
bzip2	1	1,097	11,905	1,097	1	20
gcc	1	16,697	65,519	16,697	1	187
mcf	1	1,234	12,935	1,234	1	9
gobmk	1	7,357	22,804	7,357	1	1,771
hmmer	1	1,473	16,898	1,473	1	10
sjeng	1	1,647	12,677	1,647	1	27
libquantum	1	1,184	13,405	1,184	1	9
h264ref	1	2,707	16,191	2,707	1	16
astar	1	3,398	27,301	3,398	1	29
xalancbmk	1	15,977	133,642	15,977	1	258
milc	1	1,814	14,435	1,814	1	9
namd	1	3,413	28,162	3,413	1	44
dealII	1	6,687	120,788	6,687	1	81
soplex	1	4,513	36,512	4,513	1	89
lbm	1	1,171	12,926	1,171	1	9
sphinx3	1	1,316	15,608	1,316	1	10
AIT	**1**	**4,671**	**34,692**	**4,671**	**1**	**187**

Table 3: Comparison of numbers of average indirect branches between ASLR-Guard and CFI works. The evaluations are all performed on SPEC CPU2006. AG: ASLR-Guard

in [45] is that even the AIR is 99.9%, the number of possible targets is still huge, especially for some large programs. As we can see in Table 3, ASLR-Guard only left a small portion (4%) function pointers in unsafe memory, which is much smaller than the ones of CFI implementations. Recently, some CFI variants make use of `type` to further reduce the AIT numbers [31, 42]. As shown in [31], the AIT number can be dramatically reduced to less than 1% compared with original one. So we believe AIT of ASLR-Guard can also be further reduced once we consider `type` in the future.

6.3 Performance

In this subsection, we evaluate the performance overhead of ASLR-Guard using the SPEC CPU2006 benchmarks. The results are shown in Table 4. Note that, for fairly comparison, we used the same basic compilation options (e.g., `-maccumulate-outgoing-args` and `-mno-push-args`) and only added "-aslrguard" and "-ffixed-r15" options in ASLR-Guard build. The results are the average numbers over 10 executions. On average, ASLR-Guard almost imposed no runtime (< 1%). In particular, 7 of them have even better performance than the original ones. gcc intensively calls the encryption routine of ASLR-Guard, so its performance overhead is about 10%. Compared with existing solutions for preventing code reuse attacks, ASLR-Guard is more efficient. For examples, the original CFI imposes 21%([1]), bin-CFI imposes 8.5%([45]), CCFIR imposes 3.6%([44]), readactor imposes 6.4% ([14]), and CPI imposes 8.4%([26]) overhead. Compared with safe-stack of CPI [26], performance of AG-Stack is slightly better, which is even better than the original one. With this negligible runtime overhead, we believe ASLR-Guard is a practical system to harden programs with information leaks. We also performed load-time overhead evaluation for ASLR-Guard, which is shown in Table 4. The load-time overhead is as expected, as ASLR-Guard performs remapping, relocation, and encryption in load-time. However the absolute time, i.e., 1 μs, is quite small. For space overhead, we measured the file size, which is 6% increased in ASLR-Guard. The memory space overhead has two megabytes more for safe vault.

7. DISCUSSION AND FUTURE WORK

Sophisticated attack defense. There are several sophisticated attacks proposed recently, e.g., "missing the point" [17] and COOP [35]. As the size of safe vault in ASLR-Guard is only 2MB, there is no "always allocated" memory in ASLR-Guard. Also offsets between sections remapped by ASLR-Guard are randomized. Provided these two features, "missing the point" attack is not applicable to ASLR-Guard. Regarding the COOP attack, as mentioned in §4.3, all virtual table pointers are encrypted as code locators, so attackers cannot know the locations of encrypted virtual function pointers to find vfgadgets, and thus ASLR-Guard can eliminate the COOP attack.

Reusing encrypted code locators. An astute reader may point out that in theory, all the encrypted code locators that are stored in unsafe memory can leaked and reused to construct code reuse attacks. However, to reuse these code locators, attackers have to conquer the following challenges: (1) exploiting an arbitrary read

Programs	Runtime					Load-time			Space (file size)		
	Orig (s)	AG-Stack (s)	AG-Stack Overhead	Full AG (s)	Full AG Overhead	Orig (μs)	AG (μs)	AG Overhead	Orig (KB)	AG (KB)	AG Overhead
perlbench	4.17	4.20	0.72%	4.32	3.60%	1.1	1.8	62.8%	2563	2891	12.80%
bzip2	12.2	12.3	0.82%	12.1	-0.82%	0.8	0.9	16.6%	171	182	6.43%
gcc	1.75	1.74	-0.57%	1.93	10.29%	5.5	6.5	17.9%	8174	8863	8.43%
mcf	3.14	3.10	-1.27%	3.10	-1.27%	2.4	3.2	30.8%	56	56	0.00%
gobmk	25.6	25.5	-0.39%	25.7	0.39%	3.2	5.4	67.9%	5936	6218	4.75%
hmmer	8.07	8.06	-0.12%	8.04	-0.37%	3.6	4.1	13.8%	690	710	2.90%
sjeng	5.48	5.41	-1.28%	5.56	1.46%	2.7	3.9	41.8%	285	339	18.95%
libquantum	0.161	0.165	2.48%	0.165	2.48%	2.4	2.5	4.7%	94	103	9.57%
h264ref	31.5	31.2	-0.95%	32.3	2.54%	3.0	3.8	25.6%	1487	1636	10.02%
astar	15.1	14.9	-1.32%	14.5	-3.97%	3.4	4.0	17.3%	177	181	2.26%
xalancbmk	0.592	0.600	1.35%	0.615	3.89%	9.9	10.2	2.9%	40127	40454	0.81%
milc	12.6	12.0	-4.76%	12.0	-4.76%	1.0	1.6	58.9%	317	355	11.99%
namd	22.0	21.1	-4.09%	21.1	-4.09%	3.5	5.2	45.9%	3548	3692	4.06%
dealII	73.1	76.3	4.38%	73.6	0.68%	4.1	5.1	24.2%	25001	25187	0.74%
soplex	0.070	0.071	1.43%	0.071	1.43%	3.6	5.1	42.7%	2848	2897	1.72%
lbm	3.16	3.12	-1.27%	3.16	0.00%	2.4	3.2	35.1%	53	57	7.55%
sphinx3	2.39	2.37	-0.84%	2.38	-0.42%	2.5	3.2	24.2%	469	485	3.41%
Average			**-0.33%**		**0.65 %**		**31.35%(0.86μs)**				**6.26 %**

Table 4: Evaluation for: runtime performance with only AG-Stack and full ASLR-Guard protection, load-time overhead, and increased file size.

vulnerability to leak the whole unsafe memory; (2) understanding the semantics of leaked memory, e.g., recovering object boundaries, types and which functions the leaked encrypted code locators point to; and (3) preparing parameters for the target function. Step (1) is relatively easy, if the vulnerability can be repeatedly exploited without crashing the program. However, step (2) is non-trivial. Previous works that tries to reverse engineer the data structures in a memory snapshot either require an un-randomized anchor to bootstrap the traverse [8]; or require execution traces [28]. But neither requirement is satisfiable under our attack model. Step (3) is also challenging in x86-64 platform, where parameters are passed via registers. Although a recent research has demonstrate using forged C++ object to pass parameters, as discussed in §6.2, ASLR-Guard can defeat such attack.

Despiting raising the bar for attacks, we think ASLR-Guard can be further improved in two directions: (1) reducing AIT. According to MCFI [31], binding code locator with `type` can significantly reduce the AIT to less 1% of original one. (2) code locator lifetime. We observe that many of the remaining code locators should only be dereferenced under a very specific context (e.g. pointers generated by `setjump()` should only be used in `longjump()`), and their lifetime should be limited and easy to determine.

Timing side-channel attacks. Timing attacks [17, 36] can infer the bytes at a relative address in unsafe memory or a guessed random address. ASLR-Guard does not rely on the default OS ASLR, instead it generates new random base addresses and specifies them in `mmap`, so code sections are always randomized independently, which thus has 28-bits `effective entropy` [21]. As the size of safe fault and code sections are small (say 100 MB), rewriting the data pointer with a guessed random value only has a probability of $1/2^{14}$ to hit the mapped code or safe vault. The probability can be further reduced by improving the entropy of the generated random base. Also, since all code locators in unsafe memory are encrypted, such timing attacks can only obtain the encrypted code locators. More importantly, our encryption scheme has the same computation costs for different locators, so it is resistant against timing attacks.

Dynamic code generation. Besides static code, recent research also showed the feasibility of launching code reuse attack targeting dynamically generated code [4]. Our current implementation of ASLR-Guard cannot prevent such attacks. However, we believe our current design can be naturally extended to protect dynamically generated code. Specifically, since the code cache used to store the generated code is already randomized (as code sections), we only need to identify pointers that may point to the code cache. This can be done through a combination of our memory analysis tool and analyzing the code of the JIT engine.

8. RELATED WORK

Pointer integrity. The most related work to ASLR-Guard is PointGuard [12], which also employed the idea of encrypting code pointers. However, it uses a single key to encrypt all code pointers with XORing, which is vulnerable to chosen-plaintext attack and forgery attack. Moreover, its type-based analysis cannot cover non-pointer code locators. AG-RandMap overcomes all these problems without sacrificing performance.

Code pointer integrity [26] employed type-based analysis to identify code pointers and unsafe stack objects, which misses non-pointer code locators. The 64-bit implementation of CPI also relies on randomization to protect its safe region. Because its safe region contains metadata of all recursively identified code pointers and safe stack, its size is so huge that there is an "always allocated" memory. In addition, the mapped sections in CPI have fixed offsets to each other. Due to these two issues, CPI is demonstrated to be bypassable by the "missing the point" attack [17]. On the contrary, the size of safe vault in ASLR-Guard is much smaller (2^{21} vs 2^{42}) and section-level randomization is performed 4.2. And thus "missing the point" is mitigated by ASLR-Guard.

Fine-grained randomizations. Under the standard ASLR deployed in commodity OS, the leak of a single address allows attacker to derive addresses of all instructions. To address the shortcoming, many finer-grained randomization schemes have been proposed. [22, 32] work at instruction level, [25, 43] work at basic block level, and [5] works at page level. Unfortunately, as demonstrated in [40], all these fine-grained randomization schemes are vulnerable to attacks that leverage a powerful information leak. Since ASLR-Guard aims to overcome the weakness against information leak, it is orthogonal to these work and can be deployed together to protect them from information leak based attacks.

Dynamic randomization and re-randomization. JIT-ROP [40] circumvents all fine-grained code randomizations by continually disclosing the code pages, assuming the code memory layout is fixed at runtime. To break this assumption, Isomeron [16] and Dynamic Software Diversity [13] dynamically choose targets for indirect branches at runtime to reduce the probability for adversary to predict the correct runtime address of ROP gadgets. However, the security of such scheme depends on the number of gadgets required to perform a ROP attack. If a small number (say 2) of gadgets is enough to perform the attack (e.g., calling `mprotect()`), the attack still has a high chance to survive (e.g., it is 25% in Isomeron). OS-ASR [19] prevents memory disclosures by live re-randomization. However, the effectiveness and performance are dependent on the frequency of re-randomization. For example, JIT-ROP can finish the attack as fast as 2.3s. To prevent such attack, OS-ASR need provide a re-randomization with interval less than 2.3s. Based on its evaluation, an interval with 2s will impose about 20% runtime overhead. In ASLR-GUARD, we completely prevent attackers reading the code sections and code pointers, which provides a better security. Also ASLR-GUARD imposes a negligible overhead.

Execute-only memory. Another approach to defeat information leak attacks is to combine fine-grained randomization and execute-only memory [6, 14]. However, they all require modification to the operating system. As a comparison, ASLR-GUARD does not require any OS modification thus is more practical. Moreover, leaking the address of trampoline table in Readactor may decrease the security provided by it to the one of coarse-grained CFI, as all entries in the table may be reused. To mitigate this issue, Readactor performs *register allocation randomization* and *callee-saved register reordering*. However the semantic of the whole function are remained the same. So *return-into-libc* attack still works under Readactor. ASLR-GUARD prevents all code reuse attacks, including return-into-libc.

Control flow integrity. Control flow integrity (CFI) [1] is considered as a strong defense against *all* control-flow hijacking attacks. Recent works mainly focused on making this approach more practical, eliminating the requirement of source code, reducing the performance overhead, providing modular support, or integrating into standard compiler toolchain [31, 42, 44, 45]. Unfortunately, many of these CFI implementations are bypassable [9, 15, 20]. Furthermore, CFI-like code reuse attack detections (e.g. [11, 33]) have also been found to be bypassable [15]. Compared with these works, ASLR-GUARD's encryption scheme can provide similar or better effectiveness for reducing the possible code reuse targets §6.2, and has lower performance overhead §6.3.

Type-based CFIs [2, 24, 31, 42] reduce the average indirect targets (AIT) by checking indirect branches with type. In particular, MCFI [31] checks type and number of parameters, Forward-edge CFI [42] checks the number of parameters, Safedispatch [24] and vfGuard [2] check class type for virtual function calls. All these protections are orthogonal and inspiring to ASLR-GUARD. In the future, we will also bind type to indirect branches to further reduce AIT. Opaque CFI [30] relies on static analysis to identify targets for indirect branches, and insert checks to limit each branch in the range from the lowest address to the highest address of all targets. So the security is various based on the range of bounds. Another very related work is Crypto-CFI [29] which encrypts all code pointers and return addresses. As a result, it imposes a significant overhead (45%). On the contrary, ASLR-GUARD leverages efficient protection techniques for stack and encryption, thus has a much lower performance overhead (< 1%). HAFIX [3] achieves fine-grained backward-edge CFI in hardware with a reasonable performance overhead (2%). ASLR-GUARD not only protects return address but also function pointer without the need of hardware support, and its performance is even better.

9. CONCLUSION

In this paper, we presented ASLR-GUARD, a system designed to prevent information leak from being used to bypass ASLR and launch code reuse attacks. To achieve this goal, we first conducted a systematic study on discovering all data that can be used to infer code gadgets. Then we present two techniques to protect the discovered code locator: (1) a complete decouple of code regions and data regions; and (2) novel approaches to isolate or encrypt found code locators. We have implemented a prototype of ASLR-GUARD, and our evaluation results over the prototype showed that ASLR-GUARD supports normal operations, stops code reuse attacks, and incurs very low runtime overhead (< 1% for the SPEC CPU2006 benchmarks).

10. ACKNOWLEDGMENT

We thank Tielei Wang, Laszlo Szekeres, Per Larsen and the anonymous reviewers for their helpful feedback, as well as our operations staff for their proofreading efforts. This research was supported by the NSF award CNS-1017265, CNS-0831300, CNS-1149051 and DGE-1500084, by the ONR under grant N000140911042 and N000141512162, by the DHS under contract N66001-12-C-0133, by the United States Air Force under contract FA8650-10-C-7025, by the DARPA Transparent Computing program under contract DARPA-15-15-TC-FP-006, and by the ETRI MSIP/IITP[B0101-15-0644].

References

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity. In *ACM Conference on Computer and Communication Security*, 2005.

[2] H. Y. Aravind Prakashm Xunchao Hu. vfguard: Strict protection for virtual function calls in cots c++ binaries. In *22nd Annual Network and Distributed System Security Symposium*, 2015.

[3] O. Arias, L. Davi, M. Hanreich, Y. Jin, P. Koeberl, D. Paul, A.-R. Sadeghi, and D. Sullivan. Hafix: Hardware-assisted flow integrity extension. In *52nd Design Automation Conference (DAC)*, 2015.

[4] M. Athanasakis, E. Athanasopoulos, M. Polychronakis, G. Portokalidis, and S. Ioannidis. The devil is in the constants: Bypassing defenses in browser jit engines. 2015.

[5] M. Backes and S. Nürnberger. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *23rd USENIX Security Symposium*, Aug. 2014.

[6] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny. You can run but you can't read: Preventing disclosure exploits in executable code. In *Proceedings of the 21st ACM conference on Computer and communications security*, 2014.

[7] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh. Hacking blind. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, 2014.

[8] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang. Mapping kernel objects to enable systematic integrity checking. 2009.

[9] N. Carlini and D. Wagner. Rop is still dangerous: Breaking modern defenses. In *23rd USENIX Security Symposium*, 2014.

[10] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. 2005.

[11] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng. Ropecker: A generic and practical approach for defending against ROP attacks. In *21st Annual Network and Distributed System Security Symposium*, 2014.

[12] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard TM: protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th conference on USENIX Security Symposium*, 2003.

[13] S. Crane, A. Homescu, S. Brunthaler, P. Larsen, and M. Franz. Thwarting cache side-channel attacks through dynamic software diversity. In *22nd Annual Network & Distributed System Security Symposium*, 2015.

[14] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz. Readactor: Practical code randomization resilient to memory disclosure. In *36th IEEE Symposium on Security and Privacy*, 2015.

[15] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *23rd USENIX Security Symposium*, 2014.

[16] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In *22nd Annual Network & Distributed System Security Symposium*, 2015.

[17] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi. Missing the Point(er): On the Effectiveness of Code Pointer Integrity. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2015.

[18] M. Frantzen and M. Shuey. Stackghost: Hardware facilitated stack protection. In *USENIX Security Symposium*, 2001.

[19] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, 2012.

[20] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, 2014.

[21] W. Herlands, T. Hobson, and P. J. Donovan. Effective entropy: Security-centric metric for memory randomization techniques. In *7th Workshop on Cyber Security Experimentation and Test (CSET 14)*, 2014.

[22] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson. Ilr: Where'd my gadgets go? In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, 2012.

[23] Intel. Intel 64 and ia-32 architectures software developer's manual, 2014.

[24] D. Jang, Z. Tatlock, and S. Lerner. Safedispatch: Securing C++ virtual calls from memory corruption attacks. In *21st Annual Network and Distributed System Security Symposium*, 2014.

[25] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In *Proceedings of the 22Nd Annual Computer Security Applications Conference*, 2006.

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

[27] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. Sok: Automated software diversity. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, 2014.

[28] Z. Lin, X. Zhang, and D. Xu. Automatic reverse engineering of data structures from binary execution. In *17th Annual Network and Distributed System Security Symposium*, 2010.

[29] A. J. Mashtizadeh, A. Bittau, D. Mazieres, , and D. Boneh. Cryptographically enforced control flow integrity, 2014. arXiv preprint arXiv:1408.1451.

[30] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz. Opaque control-flow integrity. In *22nd Annual Network & Distributed System Security Symposium*, 2015.

[31] B. Niu and G. Tan. Modular control-flow integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

[32] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, 2012.

[33] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent rop exploit mitigation using indirect branch tracing. In *Proceedings of the 22nd USENIX Conference on Security*, 2013.

[34] M. Prasad and T. cker Chiueh. A binary rewriting defense against stack based overflow attacks. In *In Proceedings of the USENIX Annual Technical Conference*, 2003.

[35] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *36th IEEE Symposium on Security and Privacy*, 2015.

[36] J. Seibert, H. Okhravi, and E. Soderstrom. Information Leaks Without Memory Disclosures: Remote Side Channel Attacks on Diversified Code. In *Proceedings of the 21st ACM Conference on Computer and Communications Security*, 2014.

[37] F. J. Serna. The info leak era on software exploitation, 2012. Blackhat USA.

[38] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, 2007.

[39] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, 2004.

[40] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *IEEE Symposium on Security and Privacy*, 2013.

[41] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter. Breaking the memory secrecy assumption. In *Proceedings of the Second European Workshop on System Security*, 2009.

[42] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in gcc & llvm. In *23rd USENIX Security Symposium*, 2014.

[43] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, 2012.

[44] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity and randomization for binary executables. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, 2013.

[45] M. Zhang and R. Sekar. Control flow integrity for cots binaries. In *Usenix Security*, 2013.