Practical Fine-Grained Binary Code Randomization†

Soumyakant Priyadarshan
Stony Brook University, USA
spriyadarsha@cs.stonybrook.edu

Huan Nguyen
Stony Brook University, USA
hnnguyen@cs.stonybrook.edu

R. Sekar
Stony Brook University, USA
sekar@cs.stonybrook.edu

Abstract

Despite its effectiveness against code reuse attacks, fine-grained code randomization has not been deployed widely due to compatibility as well as performance concerns. Previous techniques often needed source code access to achieve good performance, but this breaks compatibility with today's binary-based software distribution and update mechanisms. Moreover, previous techniques break C++ exceptions and stack tracing, which are crucial for practical deployment. In this paper, we first propose a new, tunable randomization technique called *LLR(k)* that is compatible with these features. Since the metadata needed to support exceptions/stack-tracing can reveal considerable information about code layout, we propose a new entropy metric that accounts for leaks of this metadata. We then present a novel metadata reduction technique to significantly increase entropy *without degrading exception handling*. This enables *LLR(k)* to achieve strong entropy with a low overhead of 2.26%.

ACM Reference Format:

Soumyakant Priyadarshan, Huan Nguyen, and R. Sekar. 2020. Practical Fine-Grained Binary Code Randomization. In *Annual Computer Security Applications Conference (ACSAC 2020), December 7–11, 2020, Austin, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3427228.3427292

1 Introduction

With widespread adoption of data execution prevention (DEP) on modern operating systems, attackers have shifted their focus from code injection to *code reuse* attacks, e.g., return-oriented programming (ROP) [47] and jump-oriented programming (JOP) [7]. Existing defenses against code reuse attacks fall into two broad categories: control-flow integrity (CFI) [2, 15, 36, 52, 61, 64] and fine-grained code randomization [6, 11, 14, 16, 18, 26, 30, 31, 38, 55, 57, 62]. Although the deterministic nature of CFI is attractive, as a code-reuse defense, CFI has a few drawbacks:

- *Use of CFI-permitted gadgets:* With CFI, attackers are unconstrained if they target "legitimate gadgets," i.e., gadgets that are reachable as per the policy enforced by CFI. In contrast, fine-grained code randomization hides the location of *every* gadget, thus requiring extra work (e.g., information leaks) before *any* of them can be used in an attack.
- *Lack of graceful degradation:* If CFI instrumentation leaves out some modules or code fragments, attackers can initiate a ROP attack from these fragments. Once initiated, such an attack is free to use *unintended gadgets* anywhere, ***including modules that have the CFI instrumentation.*** This is because CFI checks are applied only on legitimate instructions, e.g., intended returns, rather than unintended ones[1]. This contrasts with randomization, where weaknesses introduced by an unrandomized code module are limited to the gadgets within that module.
- *Compatibility:* Higher precision (aka fine-grained) CFI [15, 36, 52] suffers from compatibility problems on complex code. Coarse-grained CFI [1, 2, 61, 64] poses fewer compatibility challenges, but is more easily defeated. Code randomization typically faces far fewer compatibility problems than CFI techniques.

These factors have prompted substantial research on fine-grained code randomization. Early works [6, 30] targeted the *static ROP* threat model, where the attacker has a copy of the victim's binary code. By statically analyzing this code, he/she can identify gadgets that can be used in an attack. Code reuse attacks have since evolved to use dynamic probing of victim process code and/or data memory by leveraging memory disclosure vulnerabilities:

- *(Direct) JIT-ROP* attacks [50] rely on the identification of gadgets on the fly by disclosing victim's *code memory*.
- *Indirect disclosure ROP* attacks leak just the *data memory* — specifically, code pointers stored in data memory.

Like many recent works, we rely on execute-only code for thwarting direct JIT-ROP. The new techniques developed in this paper are thus aimed at the static ROP and indirect disclosure ROP threat models.

1.1 Motivation: Deployable Code Randomization

Despite advances in new code randomization techniques, they are not widely deployed due to several concerns described below.

Need for Source Code. Many code randomization techniques rely on a modified compiler [6, 14, 31] or special compiler options [18, 30, 57] (e.g., debug or relocation flags) that aren't enabled on production binaries. This makes them incompatible with today's dominant software deployment and update mechanisms, which involve the distribution of binary code. Even open-source software is predominantly distributed in binary format for convenience.

Performance. A low overhead is critical for the deployment of security hardening measures. Often, a 5% or lower threshold is quoted. While techniques that rely on some level of compiler support [14, 18, 30, 31, 57] have met this threshold, most binary-based techniques (e.g., [16, 26, 62]) tend to have higher overheads.

ACSAC 2020, December 7–11, 2020, Austin, USA

ACM ISBN 978-1-4503-8858-0/20/12…$15.00
https://doi.org/10.1145/3427228.3427292

[1]The attacker can use any gadget beginning in the middle of a legitimate instruction, as long as the indirect control flow instructions in the gadget are unintended.

†The first two authors contributed equally to this work, which was supported by ONR (N00014-17-1-2891). Third author's work was also supported by NSF (CNS-1918667).

Compatibility with stack tracing and C++ exceptions. A chief concern for deployed software is the support for error handling and reporting. *Unfortunately, existing fine-grained code randomization techniques don't support these features.* While this incompatibility may be acceptable for a proof-of-concept implementation, it is not a viable option for platform-wide deployment. In particular, libraries need to be compatible, or else exceptions and stack traces are broken for every application that uses them.

Some techniques (e.g., Readactor [14]) are incompatible because they violate a key assumption behind these mechanisms: that function bodies are contiguous. Many others [6, 11, 16, 18, 26, 30, 55, 57, 62] are incompatible because they fail to maintain the metadata used by these mechanisms. ***More importantly, none of the previous techniques have considered the security implications of this metadata.*** In particular, both stack tracing and exception handling operate from the "stack unwinding" information stored in the `eh_frame` section of Linux binaries. This section records the addresses of the first and last instructions of (almost) every function in the binary. *It is important to note that this information is present in stripped binaries,* and is stored in *readable memory at runtime.* Moreover, this information is not limited to C++, as stack traces are needed for C-code as well. We found that this information is present for 95% of the code on Ubuntu Linux.

Since attackers have proven adept at leaking information stored in readable memory, it is necessary to develop randomization techniques that are secure despite such leaks. In particular, many existing techniques derive the bulk of their randomness from permuting the order of functions. The availability of `eh_frame` information defeats the security of such schemes.

1.2 Approach Overview and Contributions

In this paper, we present Stony Brook Static Binary Randomizer (𝕊𝔹R) that provides the following key features:

- Compatibility with exceptions and stack traces;
- Compatibility with COTS binaries, including low-level libraries such as the system loader (`ld`) and the C-library (`glibc`);
- Support for code written in multiple languages, including C, C++, Fortran and hand-written assembly, and compiled using multiple compilers (e.g., gcc, llvm and gfortran); and
- Low runtime overhead.

𝕊𝔹R has been tested on 640MB of binaries. (This is about 2/3$^{\text{rd}}$ the size of all binaries on Ubuntu Desktop 18.04.) We plan to open-source 𝕊𝔹R in a few months. Our main contributions are as follows.

Stack-unwinding-compatible randomization. We present a new technique called *LLR(k)* that provides the following benefits:

- Each leaked code pointer reveals the locations of just k more instructions. As a result, attackers need to leak many pointers before they have sufficient gadgets for an effective payload.
- Users can easily make security vs performance trade-offs by tuning k. Larger k values yield better performance, while smaller k values offer increased security. Moreover, *LLR(k)* can be seamlessly combined with other randomization techniques.
- Our experimental results show that $k = 16$ achieves good security (in the form of high entropy) with a low overhead of 2.26%.

Unwinding Block Entropy and Reduced EH-Metadata. We show that the metadata used for C++ exceptions and stack-tracing reveals a lot of fine-grained information about instruction locations.

- We define a new entropy metric, *unwinding block entropy,* to quantify the difficulty of attacks that exploit this metadata.
- We develop a novel approach for reducing the metadata such that C++ exceptions would continue to work seamlessly, and with the same performance as before.
- We show that this metadata reduction has a major impact on our new entropy metric, increasing it by 8x.

Comparison of randomizing transformations. We present a robust implementation of 𝕊𝔹R that scales to complex binaries on 64-bit x86/Linux systems. It randomizes all code, including executables and all libraries. Using this implementation, we present a detailed experimental evaluation of the security vs performance trade-off offered by previous randomization techniques and our new *LLR(k)*.

1.3 Paper Organization

Sec. 2 provides the background on stack unwinding, our threat model, and previous randomization techniques. Our new *LLR(k)* technique is introduced in Sec. 3. A new unwinding metadata optimization is described in Sec. 4. Our new entropy metrics are presented in Sec. 5, followed by our binary instrumentation approach in Sec. 6. Implementation and evaluation are the topics of Sec. 7 and 8, followed by discussion, related work, and conclusions in Sec. 9, 10 and 11.

2 Background and Threat Model

2.1 C++ Exception and Stack Tracing Compatibility

Modern C++ compilers and runtime systems implement a "zero overhead" (aka "zero cost") exception model. This model is aimed at eliminating runtime overheads for any program that raises no exceptions, even if it includes code that uses exceptions. This is achieved by avoiding proactive book-keeping at runtime for exception handling. Instead, the compiler generates tables that include all the information necessary to process exceptions at runtime. This table is stored in read-only data sections in the binary that we will collectively refer to as EH-metadata.

On GNU/Linux, stack tracing also uses EH-metadata, so this metadata is included in code generated from many languages, including C. Even hand-written assembly in many system libraries contains EH-metadata. The vast majority of binary code on Linux systems is covered by EH-metadata — for instance, 95% of all the code in `/bin` and `/lib/x86_64-linux-gnu` on Ubuntu 18.04 Linux.

An operation central to exception processing as well as stack tracing is *stack unwinding*. This operation involves restoring the values of callee-saved registers, and restoring the stack pointer to its value when the current function was entered. On completion of unwinding, the stage is set for returning to the caller. The caller may in turn perform its own unwinding and return to its caller, and so on. For C++ programs, unwinding stops when it reaches a catch block for the current exception, or the outermost stack frame.

EH-metadata specifies: (a) the start and end locations of each function, (b) the beginning and end of each *unwinding block*, and (c)

the operations for unwinding. An unwinding block may correspond to a try-block in a C++ program, or to instructions that change the stack pointer and/or callee-saved registers. The operations for unwinding a block are usually specified as a delta over a previous unwinding block, thus revealing dependencies between them. More details on EH-metadata can be found in [37, 42].

Key Implications and Requirements for Code Randomization.

- Exception metadata needs to be updated after code movement.
- This metadata reveals a lot of information useful to attackers:
  (1) the start and end address of each function,
  (2) the start and end of each unwinding block, and
  (3) the dependence between unwinding blocks.

Our investigation shows that across a range of Linux/x86_64 binaries, ***an average function contains about a dozen unwinding blocks.*** So, unless care is taken, EH-metadata can leak a lot of information about code locations, thereby greatly degrading the effectiveness of code randomization. To address this threat, we need

- new code randomization techniques that can provide adequate security despite such leaks (Sec. 3),
- new metadata optimization techniques that minimize the amount of EH-metadata without impacting the functionality or performance of exception handling (Sec. 4), and
- new entropy metrics that assess the security provided by code randomization in the face of EH-metadata leaks (Sec. 5).

2.2 Threat Model and Security Goals

Our threat model is similar to previous work, with the key difference that attackers are aware of 𝕊𝔹ℝ's compatibility with stack traces and exceptions and hence may:

- leverage the fact that function bodies are contiguous in order to speed up their attack, and/or
- target EH-metadata specifically and disclose it. This is possible because *this metadata is present in stripped binaries,* and is stored in *readable memory at runtime.* Moreover, it typically covers 95% of all functions, including most C-code and assembly.

With these differences in mind, we outline the three threat models considered in code randomization research.

Static ROP. Although this threat model mentions ROP [47] specifically, it is intended to include other code reuse attacks that rely on existing code snippets such as JOP [7]. This threat model assumes that (a) the attacker is able to exploit a vulnerability in the victim program to hijack its control flow to start the execution of a gadget chain, and (b) the locations of these gadgets are determined on the basis of an attacker's prior knowledge of the victim program's binary. All code randomization techniques aim to take away (b), but don't always do it completely. For instance, compiler-based techniques don't randomize low-level code written in assembly.

Our goal is to defeat static ROP by ensuring that the attacker has no knowledge of *any part* of the binary code that executes at runtime, and by introducing large entropy into this binary.

JIT-ROP. The JIT-ROP threat model assumes that the victim program has a memory corruption vulnerability that provides (i) an arbitrary read capability, and (ii) an ability to hijack control-flow. It also assumes the availability of a scripting environment that (i) executes attacker-provided scripts, and (ii) can exercise these vulnerabilities. State-of-art defense against JIT-ROP relies on execute-only (i.e., non-readable) code. Since this technique imposes very low overheads and is also very strong due to its reliance on hardware memory protection, our approach will simply rely on this technique to protect against JIT-ROP. (Note that our techniques are compatible with execute-only code.)

Indirect (only) Disclosure ROP. This threat model assumes that the victim program has a memory corruption vulnerability that enables an attacker to read arbitrary memory locations. It also assumes the availability of another vulnerability that enables control-flow hijack. Finally, it assumes that code is protected from reads, so the attacker cannot use leaked pointers to search the code for usable gadgets. Instead, she targets gadgets that are adjacent to the leaked code address, or at a short distance from it.

Attackers may very well use gadgets at the leaked pointers. Preventing such reuse is hard, and is outside the scope of code randomization. Instead, our goal is to prevent attackers from using leaked pointers to identify (the locations of) *additional usable gadgets.*

The availability of ***EH-metadata greatly increases useful information that may be leaked by indirect disclosures.***

2.3 Common Randomizing Transformations

In this section, we summarize most of the fine-grained randomizing transformations that have been proposed before. These transformations proceed in two phases. The first phase determines how a function body is split into a set of partitions. In the second phase, the partitions are permuted, and jumps introduced as needed to preserve the original control-flow. Since the second phase is similar for all transformations, we focus on the first phase below.

- ***Function Reordering*** (*FR*): Proposed in the earliest works on code randomization [6, 30], this technique does not change function bodies at all — it simply permutes the order of functions in the code section. This achieves high entropy against static ROP threat model, but *FR* is insufficient if code pointers or stack-unwinding information can be leaked.
- ***ZeroJmp*** (*ZJR*): Koo et al [31] proposed to align code splits at locations terminating with unconditional jump instructions. With this alignment, *no new jumps* are introduced for randomization; instead, we simply adjust the targets of existing jumps after permuting the blocks. As a result, Koo et al achieved nearly zero overhead for this technique. We show, however, that *ZJR* is relatively weak against adversaries that can leak code pointers.
- ***Basic Block Randomization*** (*BBR*): This technique splits function bodies at basic block boundaries. A basic block is an instruction sequence with no incoming control transfers except to the first instruction, and no outgoing control transfers except through the last instruction.
- ***Pointer-Hiding Randomization*** (*PHR*): Readactor [14] introduced a *pointer hiding* defense against indirect disclosure attacks. Specifically, for every indirectly called function, they introduce a corresponding trampoline that then jumps to that function. It is only the trampoline address that is stored in memory. Since the trampoline is located randomly, it reveals no information

about possible gadgets at the beginning of the target functions. To protect return addresses, each call is replaced with a jump to a trampoline for that call-site, with the trampoline making a call to the target function. As a result, the return address only leaks the location of the trampoline.
Random placement of call-site trampolines will break stack-unwinding. So, we consider a modification of Readactor's technique that locates the trampoline at a random location within the body of the caller. In addition, code blocks between successive calls are permuted. We call this variant as *PHR*.

- ***Phantom Blocks*** (*PB*): Instead of relying purely on permutation, phantom blocks were introduced in kR^X [40] to create gaps between blocks of original code. By randomly varying the size of phantom blocks, entropy can be further increased. Moreover, these blocks can be made into "traps" by filling them with invalid code. This will cause any jumps into these blocks to terminate the victim program.
Note that *PB* does not create new splits in the function body — instead, it relies on other schemes such as *BBR* or *PHR*. Specifically, kR^X relies on the *PHR* variant described next.
- ***One-side Pointer Hiding*** (*OPHR*): Note that call-site trampolines of *PHR* require one jump into the trampoline, and a second jump out of the trampoline. Performance can be improved by removing one of these jumps. There is also a security cost, because the gadget location are hidden only on one side of the call: the side that contains a jump.

3 LLR(k): Length Limiting Randomization

Existing randomization techniques outlined above do not satisfactorily address indirect disclosure ROP that leverages EH-metadata:

- *PHR* can stop attackers from computing additional gadgets adjacent to a return address even after the attacker leaks that return address. However, if the attacker leaks a code address from EH-metadata, *PHR* cannot prevent attackers from knowing the locations of nearby gadgets.
- *ZJR* and *BBR* don't address indirect disclosures per se, but they do have a secondary effect since they chop and permute function bodies. This means that a leaked return address exposes the location of instructions in the same code block, but the gadgets in other blocks within the caller are still unknown to the attacker. Unfortunately, *ZJR* blocks can be large. In the SPEC suite, we observed thousands of blocks consisting of hundreds of instructions. Although *BBR* blocks tend to be small on average, there are still over a hundred blocks containing hundreds of instructions. In fact, we find a basic block that is 8KB-long! As a result, a leaked code address can allow an attacker to compute a large number of additional gadgets.
- *PB*s also don't address indirect disclosures, so kR^X [40] relies on *OPHR* for this purpose. Being a weaker (but faster) form of *PHR*, *OPHR* shares the weaknesses of *PHR*, i.e., no protection is offered for addresses disclosed in the EH-metadata. In addition, *OPHR* shares a drawback of *ZJR*: that the number of large *OPHR* blocks is comparable to that of *ZJR*.

In contrast, we introduce a new technique, called Length Limiting Randomization (*LLR(k)*), which limits the utility of any disclosed code address. The basic idea behind *LLR(k)* is very simple. Let s be the size of a function. We generate $p = \lfloor s/k \rfloor$ distinct random numbers $r_1, ...r_p$ over the range $[1, s-1]$. We then proceed to create a partition at each r_i. Since the number of partitions is $p+1$, the average partition size is $s/(p+1) = s/(\lfloor s/k \rfloor + 1) \leq s/(s/k) = k$.

Despite its simplicity, *LLR(k)* is quite powerful, and offers several benefits over previous techniques:

- *Tunable entropy and performance:* Small values of k mean a large number of small blocks. This increases entropy, but decreases performance because frequent jumps increase code size, while also decreasing cache locality. By the same reasoning, larger k values provide better performance while decreasing entropy.
- *Bounded utility for any disclosed address.* Since the expected length of any contiguous block of code is k, an attacker that discloses an address can expect to be able to guess the locations of up to k adjacent instructions. To access gadgets beyond this range, the attacker will have to disclose additional addresses[2].
- *Higher entropy than other techniques for the same number of partitions.* For a given average partition size, *LLR(k)* provides much higher entropy as compared to other schemes such as *ZJR* or *BBR*. For instance, consider a function of size 100 instructions, and let the average block size be 10. For this block size, both *ZJR* and *BBR* yield an entropy of 22 bits, while *LLR(k)* yields 66 bits of entropy! This is because *LLR(k)* introduces a lot of additional randomness in the placement of breaks, whereas the placement is deterministic for all other schemes discussed above.
- *Can be seamlessly combined with other randomizations.* We can start with a base randomization scheme, such as *ZJR*, *BBR*, *PHR*, or *OPHR*, and introduce additional randomness using *LLR(k)*. Suppose that the base scheme introduces breaks at $m-1$ locations, thus yielding m partitions of a function. We then eliminate these $m-1$ locations (out of a total of $s-1$ possible locations) from consideration, and number the remaining locations from 1 to $s-m$. From these $s-m$ locations, we choose $p = \lfloor s/k \rfloor - m$ random locations to create additional partitions. Note that the total number of partitions is $\lfloor s/k \rfloor$, thus ensuring the same average block size as a pure *LLR(k)* scheme.

The most obvious combination is *ZJR +LLR(k)*. In practice, there is no reason to omit *ZJR* since it has nearly zero overhead. So, we make *ZJR +LLR(k)* combination as the default, *using the term LLR(k) to refer to this combination.* Stand-alone *LLR(k)* is called pure-*LLR(k)*.

A second combination we consider is *PHR + LLR(k)*. As compared to *PHR*, we show that it provides a substantially higher entropy at a small additional performance cost.

4 Limiting Disclosures in EH-metadata

By updating EH-metadata after code randomization, the functionality of C++ exceptions and stack tracing can be restored. Unfortunately, the updated metadata reveals far too much information about the new code layout that can be leveraged to defeat randomization. Recall from Sec. 2.1 that EH-metadata reveals:

[2] Since partitions are determined by a random number generator, some *LLR(k)* partitions can be larger than k. However, unlike randomization schemes where the attacker knows the larger blocks ahead of time, the attacker cannot predict which *LLR(k)* blocks will be large. This is why we consider the expected length k as a limit on the number of gadgets an attacker can determine from a disclosed address.

(a) the start and end of each unwinding block,
(b) the dependence between successive unwinding blocks, and
(c) the operations for unwinding the stack and restoring registers.

It is easy to see that the amount of metadata is directly proportional to the number of unwinding blocks. Thus, in order to minimize disclosures through EH-metadata, we describe in Sec. 4.1 our technique for eliminating most unwinding blocks without impacting exception handling. Next, in Sec. 4.2, we discuss the spectrum of possible code transformations that preserve unwinding compatibility for the remaining blocks, and justify our specific design choice.

4.1 Reducing EH-metadata Stored in Memory

A key observation we make is that small unwinding blocks frequently consist of instructions such as `push` or `pop` that won't trigger C++ exceptions. This is because C++ exceptions are ultimately triggered by a call to a `throw` function in the standard C++ library. This means that only those unwinding blocks that contain `call` instructions can be involved in a C++ exceptions. All other unwinding blocks could only be used in stack tracing, which is typically used when a process terminates due to a fatal error. This may include the case of unhandled signals, e.g., due to memory faults, divide by zero, etc.

Based on the above observation, our design generates two versions of EH-metadata: a full version that includes all unwinding blocks, and a reduced version that only stores information for call-containing unwinding blocks. The full version is stored in a region of memory that is made unreadable, so it cannot leak to the attacker. The reduced version is the EH-metadata that is available at runtime.

When C++ exceptions occur, the above design ensures that our reduced EH-metadata will include the information needed for unwinding all the code blocks in the current call chain. Consequently, exception handling will continue to work as before.

Typically, stack tracing is invoked when a process encounters a serious error. Such an error may be detected by the program, and it may respond by calling a library function for printing the stack trace and exiting; or, it may be an unhandled error that manifests as a UNIX signal. In the former case, since a function is being invoked, all the relevant unwinding information will already be in the reduced EH-metadata. To handle the latter case, we can install a signal handler in the instrumented binary to check if the error is due to a fault triggered by an instruction execution. If so, $\mathbb{SBR}$ will replace the reduced EH-metadata with the full version. After completing its task, $\mathbb{SBR}$'s signal handler will transfer control to the application's signal handler. This kind of signal handler "hooking" can be achieved by instrumenting `glibc` functions used for signal handler registration. This is feasible since $\mathbb{SBR}$ instruments all binaries, including `glibc` and the system loader. However, we have not implemented this yet.

Note that the above design can support C++ exceptions as well stack tracing for programs written in C or other languages. We add no additional overheads for C++ exceptions, or any explicit calls to functions that perform stack-unwinding. There is additional overhead in the remaining cases, but since those cases typically occur in conjunction with process or thread termination, the additional overheads seem acceptable.

Function	**Unwinding operations for original blocks**
100:push%rbp //*Block* A_1	A_1[100-100]: RBP = *(RSP); RSP = RSP + 8
102:sub $20,%rsp//*Block* A_2	A_2[102-102]: {RSP = RSP + 20}
106:push%r8 //*Block* A_3	+ unwind operations of A_1
108:call 140	A_3[106-108]: {R8 = *(RSP); RSP = RSP + 8}
10d:pop %r8 //*Block* A_4	+ unwind operations of A_2
10f: call 120	A_4[10d-10f]: unwinding operations of A_2
114:add $20,%rsp//*Block* A_5	A_5[114-114]: unwinding operations of A_1
118:pop %rbp //*Block* A_6	A_6[118-11a]: { };
11a:ret	**Unwinding operations post-optimization**
	A_{13}[100-108]: R8 = *(RSP); RSP = RSP+28; RBP = *(RSP); RSP = RSP+8
	A_{46}[10d-11a]: RSP = RSP+20; RBP = *(RSP); RSP = RSP+8

Fig. 1: Unwinding blocks example

Fig. 1 illustrates our optimization on an example function with 6 unwinding blocks, A_1 through A_6. The second column in the figure shows the unwinding operations for A_1 to A_6. Note that the unwinding operations for A_1 undo the effect of its only instruction `push %rbp` on the stack and callee-saved registers. Unwind operations for A_2 need to undo the effect of its instruction `sub $20, %rsp` and those of the blocks that preceded it. Rather than duplicating the unwind operations of A_1 within those of A_2, a dependency on A_1 is indicated in the metadata. At runtime, the stack unwinder will observe this dependence and perform A_1's unwind operations following those of A_2. Note that the first instruction in A_4 undoes the effect of A_3 on the stack and callee-saved register. Realizing this, the compiler simply records a dependence from A_4 to the block A_2 preceding A_3.

Since A_1, A_2, A_5 and A_6 contain no calls, our optimization can delete them. In addition, we perform an additional optimization:

Expanding call-containing blocks: While unwinding blocks without calls have been removed, their presence may be partially revealed by the gaps in the ranges of remaining blocks. To avoid this, we expand call-containing blocks until they meet each other. Instead of a deterministic choice, we pick the meeting point at random so as to increase attacker effort. In the example above, A_3 has been expanded to A_{13}, and its range 100–108 combines those of A_1 to A_3. Unwinding operations from A_1, A_2 and A_3 have been consolidated in reverse order into A_{13}, ensuring the same behavior as the original code if any exception occurs within this call. A_4 has similarly been expanded to A_{46}.

4.2 Unwinding-Compatible Code Randomization

After expanding unwinding blocks as described in the previous section, the next step is to randomize the code within these blocks. We discuss two possible options in this regard and justify our choice.

Whole function randomization. This choice is motivated by the fact that the number of possible randomizations is significantly larger if we permute the whole function without placing additional constraints on the basis of unwinding blocks. Unfortunately, this increase in apparent entropy does not necessarily provide more security in our threat model. Consider two successive unwinding blocks A and B in the original code. Suppose that A is broken into fragments A_1 and A_2 and B is broken up into B_1 and B_2 and the code rearranged in the order $A_1B_1A_2B_2$, and then jumps are introduced to maintain the original control flow. Since B_1 requires a

different set of unwinding operations, it has to reside in a distinct unwinding block from A_1 and A_2. In other words, four unwinding blocks would be needed now, thus reversing the benefits of the optimization described in the last section, and exposing more information about the code layout in EH-metadata. Moreover, it is often possible to infer the dependence between A_1 and A_2 (and the lack of dependence between A_1 and B_1 or B_1 and A_2) from the associated unwinding data. *Worse, the attacker can now determine the length of blocks A_1 and A_2, thereby pinpointing the locations where the original code blocks have been partitioned.* Using dependency and block boundaries, an attacker can potentially determine the permutation that has been applied, thus negating the security benefits of randomization.

Intra-block randomization. This is the simplest option to implement because it does not change unwinding block boundaries. As such, EH-metadata remains unchanged after randomization. This implies that (a) leaks of this metadata will reveal nothing about the code randomizations performed on any block, and (b) the functionality as well as the time and space overhead of the exception handling will be exactly as before randomization. We have therefore chosen intra-block randomization in $\mathbb{SBR}$. Experimental results show that our expanded unwinding blocks are above 50% of the function size on average, so we can achieve sufficient entropy.

It should be noted that $\mathbb{SBR}$'s randomization is confined to our *expanded* unwinding blocks. As a result, they will break up some of the original unwinding blocks, e.g., A_1, A_2, etc. Hence the full EH-metadata that will be used for stack tracing can contain even more unwind blocks than the original, a factor we evaluate in Sec. 8.6.

5 Entropy: Quantifying Randomization Strength

Entropy is calculated using its information theoretic definition as $\sum_{i=1}^{n} -p_i \log p_i$, where there are n possible outcomes, with the ith outcome having a probability of p_i. (When all outcomes are equally likely, as is the case in most of our transformations, this formula simplifies to $\log n$.) As is common, we use 2 as the base for log operations below, and hence report entropy in bits. We define four distinct entropy metrics: two that have been used in previous works, and two that we introduce in this paper.

- ***Global Entropy*** (*GE*): This quantity measures the global entropy across an entire binary. If a randomization scheme can generate V distinct variants of a binary, each with a probability of $1/V$, then the *GE* of the binary is $\log V$.

  A high global entropy is an effective defense against static ROP. However, it is not a useful security measure in our threat model, where code locations may be revealed via code pointers or EH-metadata[3]. Hence we leave *GE* out of further discussion.
- ***Function Entropy*** (*FE*): This quantity measures the entropy of a single function. The *FE* of an entire binary is taken as the arithmetic mean of the *FE*'s of all the functions in the binary.

  When function bodies are contiguous, as is common with many randomization schemes, *FE* provides a good measure of security against conventional indirect disclosure attacks that leak code pointers. Although it is not as meaningful in the face of EH-metadata disclosures, we still use it in our evaluation since it is well known, and hence makes our results easier to interpret.
- ***Full Unwinding Block Entropy*** (*FUBE*): This quantity measures the entropy of a single unwinding block. It is defined similar to *FE*, but instead of applying it at the granularity of functions, we apply it at the granularity of unwinding blocks. *FUBE* represents the mean entropy across all unwinding blocks.

  FUBE targets indirect disclosure attacks that leak unmodified EH-metadata, before any of our reduction techniques (Sec. 4.1) are applied. In such a case, the attacker knows the boundaries of every unwinding block, so a randomization scheme is limited to randomizing instructions within each such block.
- ***Reduced Unwinding Block Entropy*** (*RUBE*): This quantity is similar to *FUBE*, except that it is applied to the reduced/optimized EH-metadata described in Sec. 4.1.

Our experimental results show that relatively high values of *FE* and *RUBE* can be achieved using our *LLR(k)* technique, thus showing that exception-handling compatibility does not have to come at the cost of security. Below, we outline the computation of entropy metrics for different randomization schemes.

ZJR, BBR *and* PHR. Computation of function entropy of these three methods is similar. Let m be the number of blocks after *ZJR* (or *BBR* or *PHR*) is used to partition a function. These blocks can be permuted in $m!$ ways, thus yielding an entropy of $\log m!$.

LLR(k). Recall that we apply *LLR(k)* over *ZJR*: we start with the m partitions produced by *ZJR*, and then introduce $p = \lfloor s/k \rfloor - m$ partitions using *LLR(k)*. The partitions introduced by *ZJR* are deterministic, but there is randomness in the way *LLR(k)* generates partitions. Recall that we choose these p locations out of $s - m$ possible locations, so these *LLR(k)* partitions introduce $\log \binom{s-m}{p}$ bits of entropy. In the second phase, we permute $p+m$ blocks, which yields an entropy of $\log(p+m)!$. Thus the total function entropy, in bits, is given by

$$\log \binom{s-m}{p} + \log(p+m)!$$

Our experimental results show that the first term has a substantial value, thus making our *LLR(k)* technique more effective as compared to previous techniques in terms of entropy. We can loosely view the second term as the entropy gained by "paying" the performance cost of the p newly introduced jumps. The first term can then be viewed as a "bonus" that is gained without a performance price.

For the same number of partitions, pure-*LLR(k)* will yield higher entropy than the hybrid scheme above (but may have a higher performance cost since every jump is newly introduced). This entropy can be found by setting $m = 1$ in the above formula:

$$\log \binom{s-1}{p} + \log(p+1)! = \log(p+1)(s-1)(s-2)\cdots(s-p)$$

[3] For instance, function reordering has high global entropy even for modest size binaries that have a few dozen functions. However, an attacker that leaks the location of a single instruction can immediately determine the locations of all the remaining instructions within the same function. Worse, an attacker that leaks EH-metadata (specifically, `eh_frame_hdr`) knows the location of every instruction in the binary. Thus, high global entropy means little in the context of our threat model.

6 Binary Analysis and Instrumentation

The central challenge in static binary instrumentation is that of accurately identifying code pointers. Since instrumentation typically changes code sizes, these pointers have to be "fixed up" to point to the correct post-instrumentation locations of their original targets[4]. Unfortunately, without relocation information that may not be included in COTS binaries, it is not possible to determine if a constant in a binary represents a code pointer or a data value. CCFIR [61] authors made the key observation that the widespread deployment of ASLR on Windows necessitated the inclusion of relocation information: Windows has long relied on position-dependent DLLs, so applying ASLR to DLLs involves a library transformation (called *rebasing*) that requires relocation information [33]. By leveraging this information, CCFIR achieved robust and efficient CFI instrumentation for Windows binaries.

Unix systems have long relied on position-independent libraries that can support ASLR without needing relocation information. However, on 32-bit x86 architecture, position-independence was achieved using ad-hoc, compiler-specific techniques that made it impossible to reliably identify code pointers. Moreover, executables were typically position-dependent,[5] and contained hard-coded pointers. For these reasons, approaches for static instrumentation of COTS Linux binaries often relied on address translation [49, 63, 64], a technique originally developed in dynamic binary instrumentation systems [8, 34] for runtime fix-up of code pointers. Unfortunately, address translation introduces significant complexity and runtime overhead. However, as vendors continue the push for applying ASLR to all binaries (including executables), almost all binaries on recent Unix systems have become position-independent[6]. Moreover, modern 64-bit platforms consistently use PC-relative addressing to create code pointer constants, or identify such constants using relocation information. Leveraging this, recent research [19, 41, 58] has shown that code pointers can be reliably identified and fixed up statically on these platforms. This enables fully static binary instrumentation with zero base overhead, while avoiding significant complexity that comes with address translation. The approach described below builds on these works, specifically [41].

Disassembly. Over the years, compilers on Linux have become increasingly strict about moving all data out of code segments and into a data segment. As a result, linear disassembly can achieve high accuracy [3]. Recent works such as RetroWrite [19] and Egalito [58] have also shown that complex Linux binaries can be successfully disassembled using linear disassembly.

Function Identification. Since most of our randomizing transformations operate on one function at a time, the next step is to divide the disassembled code into functions. We rely on EH-metadata to identify function boundaries. Recent work shows [42] that on Linux/x86_64 (our implementation platform), this technique is more accurate than many techniques specialized for accurate function identification [4, 5, 43, 48]. Although EH metadata may not be as complete on other platforms, this won't affect $\mathbb{SB}$R's correctness: our sole correctness-critical use of function boundaries occurs in the context of preserving the unwinding blocks that are actually present in the EH-metadata.

[4]For instance, the starting point of a function f may change from a location `0x1000` to `0x1050` after instrumentation. This requires every constant value `0x1000`, if it represents a pointer to f, to be changed to `0x1050`.
[5]This also means that they are not randomized by ASLR.
[6]About 99% of binaries on a default Ubuntu 18.04 install are position-independent.

Original code	Randomized code
1200: lea 0xf7(%rip), %rdi	L1200: lea L1300(%rip), %rdi
1209: mov -0xefe(%rip), %rax	L1209: mov L310(%rip), %rax
// load function pointer from location 30c	
120e: call *%rax	L120e: call *%rax
	jmp L1211
	// Jump table targets...
	L122a: …
	L1270: …
	L1298: …
1211: lea -0xf18(%rip),%rdi	L1211: lea L300(%rip), %rdi
1218: cmp $0x14, %rax	L1218: cmp $0x14, %rax
121c: jge 12ff	L121c: jge L12ff
121e: add %rdi,%rax	L121e: add %rdi,%rax
1221: mov (%rax), %rax	L1221: mov (%rax), %rax
1226: add %rax,%rdi	L1226: add %rax,%rdi
1229: jmp *%rdi	L1229: jmp *%rdi
// Indirect jump using jump table	
122a: · · · // code for jmp table entry 1	
1270: · · · // code for jump table entry 2	
1298: · · · // code for jump table entry 3	
12ff: ret	L12ff: ret
1300: push %rbp	L1300: push %rbp
1301: sub $0x20, %rsp	L1301: sub $0x20, %rsp
. . .	. . .
Static data:	***Static data:***
// Jump table...	// Rewritten jump table
300: 0xf2a	L300: .long L122a-L300
304: 0xf70	L304: .long L1270-L300
308: 0xf98	L308: .long L1298-L300
// Pointer constant marked for relocation	
310: 0x1500	L310: .8byte L1500

Fig. 2: Intra-function randomization. To highlight correspondence between instructions before and after randomization, (a) vertical space has been introduced to align instructions, and (b) only one code permutation is shown.

Pointer Identification and Remapping We illustrate pointer identification and remapping using the code snippet on the left of Fig. 2. Its randomized version is shown at the right of this figure.

On Linux/x86_64, there are three ways for PIC to create pointers. The first is the use of PC-relative addressing to compute the address of static data or code, e.g., the `lea 0xf7(%rip), %rdi` instruction at location 1200. This instruction moves the value 1300 into the `%rdi` register. (Note: the PC register `%rip` points to the next instruction at 1209, so $0xf7+1209 = 1300$.) The second way is by loading a pointer that is stored within the static data (or possibly the code), e.g., the instruction `mov -0xefe(%rip),%rax` at location 1209. This instruction moves the contents of location $120e - efe = 310$ into `%rax`. We identify the loaded value as a pointer because location 310 is marked for relocation. For both instructions, we ensure that the references point to the correct location after binary instrumentation by replacing the constants with labels. We use the style of BinCFI [64], where the location information produced by a disassembler (e.g., 1300) is turned into a label (e.g., $L1300$). These symbolic references are resolved by the assembler when we reassemble the randomized code shown on the right in Fig. 2.

The third way pointers are created is through pointer arithmetic. There is no need to "fix up" data pointer arithmetic: it involves adding a value to a base address, and since 𝕊𝔹ℝ does not change the distances within the data segment, there is no need to adjust this value. However, code layout is altered, so we need to adjust the new pointer value so that it accesses the same logical target as the original code. To do this, we need to know the exact value that is being added, which will be known only at runtime. Fortunately, code pointer arithmetic tends to occur only in the context of jump tables, which are typically generated by compilers from C-style switch statements. We have developed a static analysis to identify the use of jump tables and fix up the targets. This analysis is able to handle all of the binaries we have tested in our evaluation. Our analysis is similar to that of Egalito [58], so we omit a description of the analysis in order to conserve space. Instead, we illustrate how jump table accesses are processed using the example of Fig. 2.

Jump table use begins with the instruction at 1211 which loads the base address of the jump table into `%rdi` register. The index value is stored in `%rax`. This value is bounds-checked at 121c, and if this fails, the function returns by jumping to the `ret` instruction at $12ff$. Otherwise, this index is added to the base address of the jump table, and the resulting location dereferenced and loaded into `%rax` at 1221. Our analysis determines that the location dereferenced is one of 300, 304, or 308; that each of them point within the read-only data segment; and that they contain the values $f2a$, $f70$, and $f98$ respectively. Based on the instructions at 1211 and 1226, our analysis also determines that (a) these values are added to a base address 300 of the jump table, and (b) the resulting values are 122a, 1270 and 1298 respectively. It can be seen that if the jump table entries are modified as shown in Fig. 2, then the fixup will be correct. In this way, 𝕊𝔹ℝ is able to statically fixup pointer constants regardless of the manner in which they are created.

Control Flow Graph (CFG) Construction. The first step in CFG construction is to identify basic blocks, which are contiguous sequences of instructions with a single entry and a single exit. The body of the function is first broken up into blocks at control-flow transfer instructions. Since a call is a control-transfer, it terminates the current basic block, just like jumps.

Next, we break these blocks further at every control flow target. Since code pointers have been identified by now, we can introduce breaks at indirect control flow targets as well. If these breaks occur in the midst of an instruction, a disassembly error is flagged, unless they immediately follow an x86 instruction prefix, e.g., `lock`.

As the last step in CFG building, edges are created between basic blocks to capture control flow transfers. These edges encode the type of the branch instruction (conditional or unconditional, jump or call, direct or indirect, etc.) and the target (for direct transfers).

Randomizing Transformations At this point, 𝕊𝔹ℝ has all the information needed for randomizing transformations: function boundaries (for *FR* and *LLR(k)*), unwinding block boundaries (for EH-metadata-reducing transformations), and the locations of unconditional branches (for *ZJR*), basic blocks (for *BBR*), and call instructions (for *PHR* and *OPHR*). Based on this information, code is broken up at the desired locations and permuted according to the description of each of these transformations earlier in this paper.

Labels derived from the original locations of instructions are maintained during code permutation, thereby simplifying the introduction of jumps between them. For instance, in Fig. 2, a break was introduced just before the instruction at 1211, so we add a `jmp L1211` after the preceding instruction at 120e. To make it easy to see the correspondence between the original and randomized instructions, we purposely limited ourselves to a single permutation in this example, and did not reorder functions. (Code reached via the jump table has also been moved, but this does not require the introduction of additional jumps since these locations were already preceded by unconditional jumps.)

Exception handling metadata regeneration. We generate both the reduced and full unwinding information as described in Sec. 4, and then encode them into the EH-metadata sections as follows:

- `eh_frame_hdr`: This section consists of a binary search table that maps a function to its corresponding frame descriptor entry (FDE) in the `eh_frame` section. Each record of this table is pair of function start and the address of corresponding FDE. We update this information using the labels of these instructions, in the same way data values are updated using labels in Fig 2.
- `eh_frame`: This section contains the FDEs for each function. The FDE contains the function start and size which we update using labels. The FDE also contains information about each unwinding block, and the associated unwinding operations. We specify the block boundaries using labels, and have implemented an encoder for recording the unwinding operations and dependencies on preceding blocks.
- `gcc_except_table`: This section encodes the address of try blocks, the corresponding catch blocks, and any destructor calls needed (to clean up stack-allocated objects) during stack unwinding. The only change we needed to make here is to update code locations using our labels.

We used labels and assembler directives (e.g., *.byte*) to specify EH-metadata sections. This enables the randomized code with regenerated metadata to be reassembled by the system assembler. We have fully tested exception handling after this transformation.

Reassembly and ELF header update Since our transformation produces valid assembly code, as illustrated in Fig. 2, it can be assembled into an object file by the system assembler. This avoids the need to implement low-level operations, such as the computation of instruction or data offsets, in 𝕊𝔹ℝ. We then use `objcopy` to extract relevant sections of this object file and inject it into the original binary. Currently, due to some engineering limitations, we leave the original code section in its place, and add a new section with the new code. (The original code is zeroed out, so this limitation has no security impact.). We then update the ELF header to reflect the new entry point for the binary. We also update the program headers and the dynamic symbol table sections to reflect the new locations in the modified binary.

7 Implementation

𝕊𝔹ℝ implementation consists of 15.8KLoC of C++. We have developed our own ELF parsers and EH metadata decoders rather than employing pre-existing utilities. We use `objdump` for linear

Program	Exec. size (KB)	# of libs	Libr. size (MB)	Total size (MB)	Description
apt-get	43	19	9.6	9.7	Run "apt-get upgrade"
enscript	281	3	3.7	4.0	Convert text file of size > 50MB to pdf
scp	100	5	2.3	2.4	Copy 100MB file
gedit	10	110	59	59	Open, edit and save a text file
evince	442	70	30	30	Open a PDF file, view pages
gimp	6058	63	23	29	Open a JPEG image, edit(crop, blur, etc) and save. Create a drawing and save file
Wireshark	8430	77	156	164	Capture network packets for 30 mins
perl	2098	5	4	6	Run a perl script to parse a file using regex
vim	2671	14	9.6	12.3	Open a text file, edit, copy, paste, search and replace and save
vlc	14	13	6.6	6.6	Play video from a network stream
pdflatex	826	25	13.4	14.2	Compile .tex files having a total size of 100KB
Python2.7	3642	6	4	7.7	Run Pystone1.1 benchmark
tar	423	7	2.9	3.3	Compress a directory of size 3GB
Aggregate	25MB	202	197MB	222MB	

Table 3: Functionality testing on common applications

disassembly, operating on small sections at a time. As noted earlier, the implementation of signal handler hooking is not complete yet.

For jump table analysis, we first use our architecture-neutral approach for [24, 25] for lifting assembly. Our system Lisc [35] lifts assembly to an intermediate representation (IR), specifically, gcc's RTL. Jump table analysis is then performed on this IR.

For entropy caclulations, 𝕊𝔹R generates logs during the randomization process that captures detailed information such as the size and location of functions, unwinding blocks, the partition locations for *ZJR*, *BBR* and *PHR*, and the locations of any trampolines introduced. This is processed by a C++ program that contains 270 lines of entropy calculation code based on the formulas from Sec 5, and another 800 lines for input/output,

8 Experimental Evaluation

Evaluation of 𝕊𝔹R was carried out on a Ubuntu 18.04.3 system equipped with an Intel Xeon Silver 4114 2.20GHz CPU and 384GB RAM. Functionality and compatibility tests were performed on a collection of frequently used applications and the SPECspeed 2017 benchmark suite. In-depth evaluation of performance and security of *ZJR*, *BBR*, *PHR*, *LLR(k)* and *PHR +LLR(k)* were based on the SPEC suite.[7] For measuring security, we used *FE*, *FUBE* and *RUBE*.

[7]We omitted *FR* and *PB* randomization techniques because they do not, by themselves, address indirect disclosures; and *OPHR* because of its similarity to *PHR*.

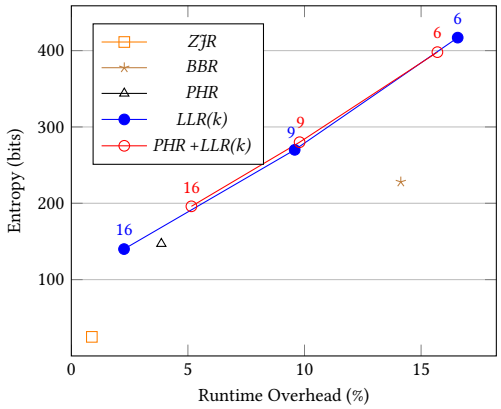

Fig. 4: Function Entropy vs Runtime Overhead (SPECspeed 2017)

8.1 Functionality Evaluation

Low-level Libraries. We randomized glibc (`libc-2.27.so`), the loader (`ld-2.27.so`) and libpthread (`libpthread-2.27.so`). They contain about 2.3MB of low-level code, with significant amount of hand-written assembly. *We replaced these standard libraries with their randomized versions and rebooted the system, and verified that the system started up properly.* We used a variety of command-line and graphical applications and verified that they worked as expected. These tests were repeated for all randomization techniques. Note that every application on the system was using 𝕊𝔹R-randomized versions of these libraries, but the application code was not randomized in this test. Tests involving application code are described below.

Commonly Used Applications. We have tested 𝕊𝔹R with many common applications shown in Table 3. The table shows the application name, the size of the executable, the number and aggregate sizes of libraries used by the executable, etc. It also describes the test performed to check its functionality. *Altogether, these tests required the transformation of 197MB of binaries contained in 202 shared libraries,* as shown in Table 9 on Page 14.

SPECspeed 2017. This benchmark consists of 19 programs[8] in 3 languages: C, C++ and Fortran. We compiled these programs using gcc, llvm and gfortran. The total size of these binaries was 420MB. We verified that the randomization techniques discussed in Sec. 2.3 and Sec. 3 preserve the functionality of all the resulting binaries, i.e., they continue to produce the correct results. Two of these programs, `omnetpp` and `leela`, use exceptions, and they continued to work correctly with our reduced metadata.

8.2 Performance vs Security Trade-off

Fig. 4 plots the entropy against the runtime overhead of various randomization techniques. While this chart is based on function entropy, a chart based on RUBE, which measures resistance against EH-metadata-aware attacks, is very similar. (See Fig. 8 on Page 14.) Deterministic techniques such as *ZJR*, *BBR* and *PHR* represent single points in this graph. But since *LLR(k)* and *PHR +LLR(k)* provide a

[8]The benchmark contains 20 programs, but we found that one of them (`cam4`) always exits with a segmentation fault. We have not been able to determine the cause, but since the problem occurs with the base version, before any processing by 𝕊𝔹R, we excluded it in our experiments.

tunable parameter k, we can obtain different entropies at different performance costs.

From the chart, it is clear that neither *ZJR* nor *BBR* is an attractive choice for deployment. *ZJR* sports a low 1% overhead, but its low entropy, at about one-sixth of *LLR(16)*'s, makes it vulnerable in our threat model. While *BBR* offers a high entropy, this comes at a steep 14% overhead. From the chart, we can see that *LLR* can match *BBR*'s entropy at just half its overhead. Alternatively, *LLR* can be tuned to match the performance of *BBR* while providing 60% more entropy.

LLR(16) provides a good combination of per-function entropy (140 bits average) and low overhead (2.26%) across the SPEC suite. *PHR*'s entropy is very close to *LLR(16)*'s, but it has a 70% higher overhead than *LLR(16)*.

Where security is a priority, *PHR +LLR(16)* is an excellent choice. It deterministically protects all code pointers in data, and in addition, offers the assurance of *LLR(16)* that each EH-metadata leak will reveal the location of at most 16 other instructions. Its overhead is not negligible, but 5% is acceptable in many settings.

8.3 Function Entropy

Although *ZJR* has very low overhead, its *FE*, shown in Table 5, is way too low — just 25 bits. This makes it vulnerable to indirect disclosure attacks.

BBR provides the highest *FE* of any randomization technique discussed so far, at 228 bits. However, as discussed before, *LLR* as well as *PHR +LLR* can achieve a better combination of entropy and performance.

PHR ensures that leaked pointers, including function pointers and return addresses, don't reveal the locations of instructions adjacent to the pointer. However, in a randomization scheme that keeps function bodies together, this is not sufficient to prevent the attacker from accessing other instructions in the same function. Thus, the only protection comes in the form of entropy — making it difficult to predict the instruction that the attacker is able to access. *PHR*'s entropy of 147 bits is quite good (and comparable to *LLR(k)*).

PHR +LLR(16) offers a 40% improvement in entropy over *PHR* at a 33% higher overhead. From Fig. 4, it is easy to see that this technique offers the best combination of security strength and performance in contexts where overheads $\geq$ 5% are acceptable.

8.4 Full & Reduced Unwind Block Entropy (*FUBE* & *RUBE*)

While *FE* has the benefit of familiarity, it is not very useful in our threat model where the attacker can target EH-metadata and stack-unwinding-compatibility. In Sec. 5, we developed two new metrics, *FUBE* and *RUBE*, for this specific purpose. We use them to evaluate *BBR*, *PHR*, *LLR(k)* and *PHR +LLR(k)* below.

FUBE. This metric captures the difficulty of attacks when all of the EH-metadata generated by the compiler is included in the binary. As shown in Table 6, *FUBE* is very low across the board for all randomization schemes. In other words, it is not feasible to develop a secure randomization scheme if the full unwinding information generated by the compiler is left in the binary.

RUBE. This metric captures the difficulty of carrying out attacks after the metadata reduction and optimization techniques described in Sec. 4 have been applied. *RUBE* values in Table 6 show a dramatic improvement over that of *FUBE*. In fact, they are about half of the *FE* values shown in Table 5. This is because the techniques of Sec. 4 were able to remove 85% of all the unwinding blocks that were originally present. This translates to a 6.7x reduction in the number of unwinding blocks. As a result, there are just under 2 unwinding blocks per function on average, as compared with 13 generated by the compiler. It can be seen from the entropy formulas in Sec. 5 that the entropy increases roughly linearly with the size s of the entity being randomized. Given that unwinding blocks, which are the entities being randomized, are about half the size of functions, it is understandable that *RUBE* is roughly half of *FE*.

Program	*ZJR*	*BBR*	*PHR*	*LLR(16)*	*PHR + LLR(16)*
perlbench	35	246	148	58	150
gcc	31	262	178	55	179
bwaves	4	60	64	165	159
mcf	11	81	42	29	44
cactuBSSN	71	407	279	149	312
lbm	5	24	21	42	47
omnetpp	13	61	71	16	72
wrf	34	442	333	446	492
xalancbmk	20	102	92	26	92
x264	18	154	87	88	108
pop2	14	234	206	168	238
deepsjeng	20	159	81	54	87
imagick	27	255	215	81	221
leela	28	144	146	51	149
nab	17	145	103	61	112
exchange2	80	804	149	375	352
fotonik3d	18	394	301	438	489
roms	28	298	244	332	384
xz	10	61	38	21	43
Mean	**25**	**228**	**147**	**140**	**196**

Table 5: Function Entropy on SPECspeed 2017

Fig. 8 on Page 14 provides a way to compare different randomization techniques. It is qualitatively similar to Fig. 4 that is based on *FE*. Thus, we can make the following observations: (a) *LLR(k)* can be tuned to provide the same entropy as *BBR* at about half the performance cost, or about 2x the entropy at the same cost, (b) *PHR +LLR(k)* provides the best combination of security and performance in contexts where a runtime overhead $\geq$ 5% is acceptable.

8.5 Runtime Overhead

Table 7 compares the runtime overhead of *LLR(k)* with previous code randomization techniques. Each SPECspeed binary was randomized with 5 distinct random seeds. Each randomized variant was run 5 times, and the average of the medians for each variant was taken as the runtime of a randomized binary.

Runtime overhead for a randomized executable can be attributed to (i) additional jump instructions introduced and (ii) negative effect of code reordering on cache locality. Since *ZJR* doesn't introduce any new jumps, its overhead must purely be from cache locality effects. Almost every binary in the table has close to zero overhead for *ZJR* except xalancbmk at 6%. CCR [31] also reports a 5% overhead on this benchmark.

Program	Full Metadata					Reduced Metadata				
	ZJR	*BBR*	*PHR*	*LLR(16)*	*PHR + LLR(16)*	*ZJR*	*BBR*	*PHR*	*LLR(16)*	*PHR + LLR(16)*
perlbench	2	17	10	4	10	12	93	55	24	55
gcc	3	23	15	5	16	17	149	100	31	101
bwaves	0	4	4	11	11	0	43	42	127	125
mcf	1	7	4	3	4	5	58	30	22	32
cactuBSSN	5	28	18	11	21	39	215	140	83	160
lbm	1	4	3	7	8	1	15	12	34	35
omnetpp	1	6	7	2	7	8	38	44	10	44
wrf	1	11	7	11	12	11	147	105	145	165
xalancbmk	2	12	11	3	11	15	77	69	20	69
x264	1	13	7	7	9	11	95	55	53	69
pop2	1	10	9	7	11	4	77	67	58	83
deepsjeng	2	14	7	5	8	11	92	50	32	52
imagick	1	13	9	4	10	6	62	46	22	49
leela	3	15	16	6	16	21	118	123	48	126
nab	1	11	8	5	9	7	72	52	34	58
exchange2	8	83	15	40	37	66	812	167	439	401
fotonik3d	1	27	21	32	37	8	241	190	289	331
roms	1	12	10	13	18	10	107	92	119	151
xz	1	6	3	2	4	6	38	22	14	25
Mean	**2**	**17**	**10**	**9**	**14**	**13**	**134**	**77**	**84**	**112**

Table 6: Full & Reduced Unwind Block Entropy (*FUBE* & *RUBE*) on SPEC 2017.

Program	*ZJR*	*BBR*	*PHR*	*LLR(16)*	*PHR + LLR(16)*
perlbench	0.3%	56.7%	15.5%	1.2%	14.5%
gcc	3.2%	35.5%	13.3%	4.8%	13.8%
bwaves	-0.5%	-0.4%	-0.4%	-0.6%	-1.0%
mcf	1.1%	17.4%	5.1%	1.7%	6.0%
cactuBSSN	1.3%	3.8%	2.4%	3.5%	4.6%
lbm	0.0%	-2.1%	-1.0%	-3.0%	-1.8%
omnetpp	1.0%	10.4%	5.0%	2.3%	4.3%
wrf	0.0%	3.2%	0.5%	1.3%	0.8%
xalancbmk	6.0%	35.4%	10.2%	6.5%	9.7%
x264	-1.5%	9.6%	0.0%	9.1%	12.2%
pop2	2.2%	10.0%	2.2%	5.3%	3.7%
deepsjeng	0.1%	23.6%	8.5%	2.4%	10.2%
imagick	0.1%	11.1%	0.3%	1.6%	0.6%
leela	-0.4%	20.0%	10.4%	2.3%	11.8%
nab	0%	3.0%	0.3%	1.0%	0.6%
exchange2	1.5%	54.3%	1.8%	12.9%	12.2%
fotonik3d	0.7%	-1.4%	0.6%	-1.1%	1.1%
roms	0.0%	0.3%	-0.2%	0.2%	0.0%
xz	0.2%	6.2%	0.7%	2.2%	2.0%
Geo. Mean	**0.88%**	**14.13%**	**3.86%**	**2.26%**	**5.14%**

Table 7: Runtime overhead on SPECspeed 2017 benchmark suite.

BBR incurs a significant 14.13% overhead because (a) it introduces many new jumps, and (b) the cache effects of permuting at much finer granularity than *ZJR* will correspondingly be larger. One factor in this high overhead is that we treat a call as an end of a basic block, which may not be the case in alternative *BBR* implementations.

PHR is implemented on top of *ZJR*. Of the trampolines added by *PHR* (see Sec. 2.3), the ones with the most performance impact are the two jumps surrounding each call. As a result of these, programs that make frequent calls can have overheads as high as 15%. The average is a moderate 3.86% overhead.

LLR(16) is also implemented on top of *ZJR*, and its overhead is proportional to the additional partitions that it introduces. Although there are two benchmarks with 9% or slightly higher overheads, the average is a relatively low 2.26%.

Since *PHR +LLR(16)* introduces more partitions than either *PHR* or *LLR(16)*, we expect its overhead to be higher than both. In fact, the overhead of *PHR +LLR(16)* tends to be close to the maximum of the *PHR* and *LLR(16)*, with an average close to 5%.

8.6 Memory Overhead

Our approach of intra-block randomization does not change the number of unwinding blocks or the data associated with them, and hence should have zero space overhead. However, in practice, our implementation uses labels, and cannot encode constants into the smallest number of bytes. This results in a 13.8% overhead when re-creating the EH-metadata. With more engineering effort, this can be brought down to zero, but we have not pursued this because our main focus is on the size *after* the metadata reduction techniques of Sec. 4 have been applied. We find that after the reduction, EH-metadata has shrunk to 50% of the original size. (Although the number of unwinding blocks have been decreased by more than 6x, the reduction in metadata size is more modest. This is because, as illustrated in Fig. 1, the unwind data for the merged blocks tends to accumulate much of the data from the original blocks.)

We also need to generate the full metadata that will be used for stack tracing on faults. As discussed before, because we have expanded call-containing blocks into nearby blocks that don't contain calls, and permuted these merged blocks, we have effectively done something similar to whole function randomization: we have chopped up existing unwinding blocks into pieces and permuted them. As a result, there are many more unwinding blocks in this case, so the metadata increases by 45%.

9 Discussion

Code Signing. Linux distributions verify code signatures at the time of software installation and updates. Our system performs its randomization on the installed (or patched) versions, and hence does not interfere in any way with current distribution models. This same comment applies to software updates and patches as well: signature checking is performed on the update, and after that, the concerned binaries are updated. 𝕊𝔹R can then randomize these updated binaries, thus making it compatible with prevalent software distribution and update mechanisms on Linux.

Rerandomization. Our system can support periodic rerandomization of binaries on the disk. Such rerandomization may be initiated on a regular basis, e.g., every few days. Alternatively, it may be triggered after a binary has been loaded a certain number of times.

COOP and AOCR. Code randomization techniques excel at stopping attacks that access code snippets that won't be used by legitimate code. Stopping attacks that use legitimate targets, such as entire functions, is much harder. 𝕊𝔹R can prevent control-flow hijacks that employ whole function code reuse only to the extent

that the attacker does not know the function's location. However, if the attacker can find its location through leaked pointers, then attacks that reuse the target function cannot be stopped by SBR. Hence SBR, like previous code randomization techniques, is vulnerable to counterfeit object oriented programming (COOP) [46] and address-oblivious code reuse (AOCR) [45] attacks.

10 Related work

Control Flow Integrity. Control flow integrity (CFI) [1, 2] techniques monitor indirect control flow transfers, permitting only those that are consistent with a statically inferred control-flow graph. They provide a principled foundation for building other security mechanisms such as software fault isolation [54, 59] and other forms of policy enforcement [21, 65]. However, as mentioned in the introduction, they have several weaknesses as a defense against code reuse attacks, and have been shown to be vulnerable [10, 46]. Coarse-grained CFI techniques are particularly vulnerable, while fine-grained techniques tend to be less compatible. To address compatibility, researchers have focused on solutions that target specific code pointer types such as those used in C++ virtual calls [23, 60] and returns [9, 17, 44]. There have also been recent works [20, 28, 29, 39, 53] utilizing hardware features for performance.

Code Randomization. Since its introduction by Bhatkar et al [6], fine-grained code randomization has been the focus of numerous research efforts over the past fifteen years [11, 13, 14, 16, 18, 26, 27, 30–32, 38, 55, 57, 62]. Earlier techniques [6, 13, 18, 26, 27, 30, 32, 38, 55] were focused on the static-ROP threat model. More recent techniques (e.g., [11, 14, 16, 57, 62]) address JIT-ROP and indirect disclosure based ROP, as discussed below.

Isomeron develops a defense against JIT-ROP attacks that relies on randomly switching between two copies of a program's code at runtime, while ensuring that calls from one copy return to the same copy. The mechanism for ensuring this is similar to shadow stack, with its potential for impacting compatibility. More important, the content of the shadow stack needs to be protected from disclosures.

Rather than protecting code pointers from being leaked, Secret [62] leverages its use of runtime code pointer translation to make leaked pointers useless. In particular, this translation can incorporate a random permutation of the code space, thereby destroying any relationship between a leaked pointer and the locations of nearby gadgets. Unfortunately, address translation imposes significant overhead. CodeArmor [11] reduces this overhead by using a random linear offset for translation, instead of the hashtable needed for a permutation. However, this makes the method susceptible to attacks that infer this random value. This is mitigated by CodeArmor's ability for runtime re-randomization.

Shuffler [57] is another technique that implements runtime re-randomization. It only adds redirection to indirect jump/call targets while relying on heuristics to protect return addresses. While Shuffler required compiler help to randomize binaries, Egalito [58] eliminates this requirement for x86_64 binaries.

Whereas the above techniques use a combination of techniques to protect against code disclosures, recent works have gravitated towards execute-only (i.e., non-readable) code [12, 22, 51, 56] for defending against JIT-ROP attacks. Since this technique imposes very low overheads and is also very strong due to its reliance on hardware memory protection, we will also rely on existing implementations of this mechanism for JIT-ROP defense.

Readactor [14] is a comprehensive compiler-based mitigation for code reuse attacks. As noted earlier, *PHR* is a stack-unwinding-compatible version of their pointer hiding. Our *PHR* implementation protects all the pointers protected by Readactor, but unlike Readactor, we do not require source code. Their performance overhead of 4.6% (which includes 0.5% for code-data separation and 4.1% for pointer hiding reported [14]) is a bit higher than our overhead of 3.86% for *PHR*, but a direct comparison is not possible because they use SPEC 2006 vs our SPEC 2017. Their design can also offer higher entropy (as the trampolines can be located far from the rest of program code) at the expense of breaking C++ exceptions and stack tracing. Our design maintains compatibility with these features, while still achieving a high average function entropy of 147 bits.

kR$\widehat{}$X [40] is a compiler-based defense that combines code diversification with execute-only memory and other techniques in order to thwart JIT-ROP in kernel code. Their phantom blocks idea, discussed in Sec. 2.3, can provide an additional improvement to the entropy of our *LLR(k)*. However, we have not considered it in our implementation because phantom blocks do not directly address indirect disclosures, and moreover, have a significant memory cost.

CCR is a hybrid approach to achieve fine-grained randomization at a low performance overhead. It includes (a) a compiler plugin to extract metadata, and (b) a static binary rewriter. This hybrid approach maintains compatibility with prevalent software distribution models, while avoiding the high overhead associated with most previous techniques that offered a similar level of compatibility. SBR is able to achieve its performance without compiler help.

We make several new contributions over all the above works. Ours is the first work to systematically study how EH-metadata can undermine code randomization, and to propose a secure code randomization defense that is compatible with exceptions and stack-tracing. Our technique offers low performance overheads while operating on COTS binaries. Moreover, they can be tuned to achieve a range of security and performance goals.

11 Conclusions

In this paper, we presented SBR, a new approach for fine-grained code randomization. By operating on COTS binaries, our technique maintains full compatibility with today's software distribution and patching mechanisms. Unlike previous works, our approach is compatible with C++ exceptions and stack tracing, two features that are crucial for deployment. We show that the metadata needed by these features can be abused by attackers. We presented several new techniques that, together, achieve fine-grained code randomization that is robust in this threat model, and achieves excellent performance. We expect to open-source SBR in the coming months. Our experimental results show that SBR offers a compelling combination of features, making it suitable for deployment.

References

[1] Martın Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2005. CFI: Principles, implementations, and applications. In *ACM CCS*.

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *ACM TISSEC* (2009).

[3] Dennis Andriesse, Xi Chen, Victor Van Der Veen, Asia Slowinska, and Herbert Bos. 2016. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In *USENIX Security Symposium*.

[4] Dennis Andriesse, Asia Slowinska, and Herbert Bos. 2017. Compiler-agnostic function detection in binaries. In *IEEE European Symposium on Security and Privacy*.

[5] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley. 2014. BYTEWEIGHT: Learning to Recognize Functions in Binary Code. In *USENIX Security*.

[6] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. 2005. Efficient techniques for comprehensive protection from memory error exploits. In *USENIX Security Symposium*.

[7] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. 2011. Jump-oriented programming: a new class of code-reuse attack. In *ASIACCS*.

[8] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. 2003. An infrastructure for adaptive dynamic optimization. In *Code Generation and Optimization*.

[9] Nathan Burow, Xinping Zhang, and Mathias Payer. 2019. SoK: Shining light on shadow stacks. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 985–999.

[10] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. 2015. Control-flow bending: On the effectiveness of control-flow integrity. In *USENIX Security Symposium*.

[11] Xi Chen, Herbert Bos, and Cristiano Giuffrida. 2017. CodeArmor: Virtualizing the code space to counter disclosure attacks. In *Euro S&P*.

[12] Yaohui Chen, Dongli Zhang, Ruowen Wang, Rui Qiao, Ahmed Azab, Long Lu, Hayawardh Vijayakumar, and Wenbo Shen. 2017. NORAX: Enabling Execute-Only Memory for COTS Binaries on AArch64. In *IEEE Security and Privacy*.

[13] Mauro Conti, Stephen Crane, Tommaso Frassetto, Andrei Homescu, Georg Koppen, Per Larsen, Christopher Liebchen, Mike Perry, and Ahmad-Reza Sadeghi. 2016. Selfrando: Securing the tor browser against de-anonymization exploits. *Proceedings on Privacy Enhancing Technologies* (2016).

[14] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. 2015. Readactor: Practical code randomization resilient to memory disclosure. In *IEEE Security and Privacy*.

[15] Lucas Davi, Ra Dmitrienko, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund, Stefan Nürnberger, and Ahmad reza Sadeghi. 2012. MoCFI: a framework to mitigate control-flow attacks on smartphones. In *NDSS*.

[16] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z Snow, and Fabian Monrose. 2015. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In *NDSS*.

[17] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. 2011. ROPdefender: a detection tool to defend against return-oriented programming attacks. In *ASIACCS*.

[18] Lucas Vincenzo Davi, Alexandra Dmitrienko, Stefan Nürnberger, and Ahmad-Reza Sadeghi. 2013. Gadge me if you can: secure and efficient ad-hoc instruction-level randomization for x86 and ARM. In *ACM CCS*.

[19] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. 2020. RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization. In *IEEE Symposium on Security and Privacy*.

[20] Ren Ding, Chenxiong Qian, Chengyu Song, Bill Harris, Taesoo Kim, and Wenke Lee. 2017. Efficient protection of path-sensitive control security. In *USENIX Security Symposium*.

[21] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C Necula. 2006. XFI: Software guards for system address spaces. In *Operating systems design and implementation*.

[22] Jason Gionta, William Enck, and Peng Ning. 2015. HideM: Protecting the Contents of Userspace Memory in the Face of Disclosure Vulnerabilities. In *Data and Application Security and Privacy (CODASPY)*.

[23] Istvan Haller, Enes Göktaş, Elias Athanasopoulos, Georgios Portokalidis, and Herbert Bos. 2015. Shrinkwrap: Vtable protection without loose ends. In *ACSAC*.

[24] Niranjan Hasabnis and R Sekar. 2016. Extracting Instruction Semantics Via Symbolic Execution of Code Generators. In *ACM Foundations of Software Engineering*.

[25] Niranjan Hasabnis and R Sekar. 2016. Lifting assembly to intermediate representation: A novel approach leveraging compilers. In *Architectural Support for Programming Languages and Operating Systems*.

[26] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W Davidson. 2012. ILR: Where'd my gadgets go?. In *IEEE Security and Privacy*.

[27] Andrei Homescu, Steven Neisius, Per Larsen, Stefan Brunthaler, and Michael Franz. 2013. Profile-guided automated software diversity. In *CGO*.

[28] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R Harris, Taesoo Kim, and Wenke Lee. 2018. Enforcing unique code target property for control-flow integrity. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*.

[29] Mustakimur Rahman Khandaker, Wenqing Liu, Abu Naser, Zhi Wang, and Jie Yang. 2019. Origin-sensitive control flow integrity. In *USENIX Security Symposium*.

[30] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. 2006. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *Annual Computer Security Applications Conference*.

[31] Hyungjoon Koo, Yaohui Chen, Long Lu, Vasileios P Kemerlis, and Michalis Polychronakis. 2018. Compiler-assisted code randomization. In *Security and Privacy*.

[32] Hyungjoon Koo and Michalis Polychronakis. 2016. Juggling the gadgets: Binary-level code randomization using instruction displacement. In *Asia CCS*.

[33] Lixin Li, Jim Just, and R. Sekar. 2006. Address-space randomization for windows systems. In *Annual Computer Security Applications Conference*.

[34] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Programming language design and implementation*.

[35] Huan Nguyen, Niranjan Hasabnis, and R Sekar. 2019. LISC v2: Learning Instruction Semantics from Code Generators. http://www.seclab.cs.sunysb.edu/seclab/liscV2/. Accessed: 2019-06-03.

[36] Ben Niu and Gang Tan. 2014. RockJIT: Securing just-in-time compilation using modular control-flow integrity. In *ACM CCS*.

[37] James Oakley and Sergey Bratus. 2011. Exploiting the Hard-Working DWARF: Trojan and Exploit Techniques with No Native Executable Code. In *WOOT*.

[38] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. 2012. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Security and Privacy*.

[39] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. 2013. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *USENIX Security*.

[40] Marios Pomonis, Theofilos Petsios, Angelos D Keromytis, Michalis Polychronakis, and Vasileios P Kemerlis. 2017. kRˆ X: Comprehensive kernel protection against just-in-time code reuse. In *EuroSys*.

[41] Soumyakant Priyadarshan. [n.d.]. A Study of Binary Instrumentation Techniques. Research Proficiency Report, Secure Systems Lab, Stony Brook University, http://seclab.cs.sunysb.edu/seclab/pubs/soumyakant_rpe.pdf. Accessed: 2020-08-30.

[42] Soumyakant Priyadarshan, Huan Nguyen, and R. Sekar. 2020. On the Impact of Exception Handling Compatibility on Binary Instrumentation. In *ACM FEAST*.

[43] Rui Qiao and R Sekar. 2017. A Principled Approach for Function Recognition in COTS Binaries. In *Dependable Systems and Networks (DSN)*.

[44] Rui Qiao, Mingwei Zhang, and R Sekar. 2015. A Principled Approach for ROP Defense. In *Annual Computer Security Applications Conference*.

[45] Robert Rudd, Richard Skowyra, David Bigelow, Veer Dedhia, Thomas Hobson, Stephen Crane, Christopher Liebchen, Per Larsen, Lucas Davi, Michael Franz, et al. 2017. Address Oblivious Code Reuse: On the Effectiveness of Leakage Resilient Diversity. In *NDSS*.

[46] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *IEEE Security and Privacy*.

[47] Hovav Shacham et al. 2007. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *ACM CCS*.

[48] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. 2015. Recognizing functions in binaries with neural networks. In *USENIX Security Symposium*.

[49] Matthew Smithson, Khaled ElWazeer, Kapil Anand, Aparna Kotha, and Rajeev Barua. 2013. Static binary rewriting without supplemental information: Overcoming the tradeoff between coverage and correctness. In *Working Conference on Reverse Engineering (WCRE)*.

[50] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2013. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *IEEE Security and Privacy*.

[51] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. 2015. Heisenbyte: Thwarting memory disclosure attacks using destructive code reads. In *ACM CCS*.

[52] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-Edge Control-Flow Integrity in GCC and LLVM. In *USENIX Security*.

[53] Victor Van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. 2015. Practical context-sensitive CFI. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 927–940.

[54] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient software-based fault isolation. In *SOSP*.

[55] Richard Wartell, Vishwath Mohan, Kevin W Hamlen, and Zhiqiang Lin. 2012. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *ACM CCS*.

[56] Jan Werner, George Baltas, Rob Dallara, Nathan Otterness, Kevin Z Snow, Fabian Monrose, and Michalis Polychronakis. 2016. No-execute-after-read: Preventing code disclosure in commodity software. In *ASIACCS*.
[57] David Williams-King, Graham Gobieski, Kent Williams-King, James P Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P Kemerlis, Junfeng Yang, and William Aiello. 2016. Shuffler: Fast and deployable continuous code re-randomization. In *OSDI*.
[58] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P Kemerlis. 2020. Egalito: Layout-Agnostic Binary Recompilation. In *ASPLOS*.
[59] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *IEEE Symposium on Security and Privacy*.
[60] Chao Zhang, Chengyu Song, Z. Kevin Chen, Zhaofeng Chen, and Dawn Song. 2015. VTint: Protecting Virtual Function Tables' Integrity. In *NDSS*.
[61] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. Practical control flow integrity and randomization for binary executables. In *IEEE Security and Privacy*.
[62] Mingwei Zhang, Michalis Polychronakis, and R Sekar. 2017. Protecting COTS Binaries from Disclosure-guided Code Reuse Attacks. In *Annual Computer Security Applications Conference*.
[63] Mingwei Zhang, Rui Qiao, Niranjan Hasabnis, and R Sekar. 2014. A platform for secure static binary instrumentation. *ACM VEE* (2014).
[64] Mingwei Zhang and R Sekar. 2013. Control flow integrity for COTS binaries. In *USENIX Security*.
[65] Mingwei Zhang and R Sekar. 2015. Control flow and code integrity for COTS binaries: An effective defense against real-world ROP attacks. In *ACSAC*.

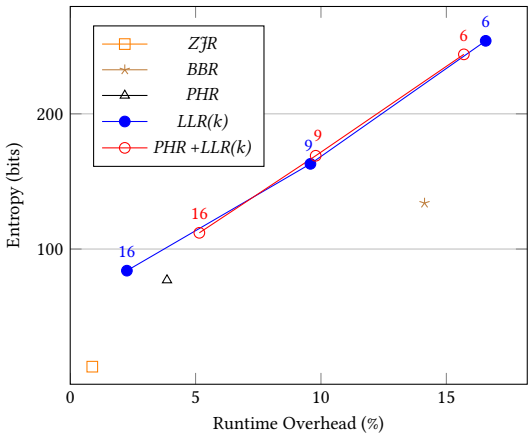

Fig. 8: Reduced Unwinding Block Entropy vs Runtime Overhead (SPECspeed 2017)

Module name	Size
libwireshark.so.11.1.10	77.95MB
libicudata.so.60.2	25.66MB
libgtk-3.so.0.2200.30	7.01MB
libQt5Core.so.5.9.5	5.28MB
libpython3.6m.so.1.0	4.47MB
libgtk-x11-2.0.so.0.2400.32	4.25MB
libicui18n.so.60.2	2.63MB
libpoppler.so.73.0.0	2.58MB
libc-2.27.so	1.94MB
libxml2.so.2.9.4	1.75MB
libapt-pkg.so.5.0.2	1.74MB
libicuuc.so.60.2	1.71MB
libm-2.27.so	1.62MB
libgio-2.0.so.0.5600.4	1.61MB
libQt5Network.so.5.9.5	1.54MB
libstdc++.so.6.0.25	1.52MB
libunistring.so.2.1.0	1.49MB
libnss3.so	1.26MB
libgstreamer-1.0.so.0.1405.0	1.23MB
libX11.so.6.3.0	1.22MB
libp11-kit.so.0.3.0	1.18MB
libcairo.so.2.11510.0	1.11MB
libQt5Multimedia.so.5.9.5	1.09MB
libglib-2.0.so.0.5600.4	1.09MB
libvlccore.so.9.0.0	1.05MB
libepoxy.so.0.0.0	1MB
libgdk-3.so.0.2200.30	0.96MB
libgedit.so	0.87MB
libkrb5.so.3.3	0.84MB
libspandsp.so.2.0.0	0.77MB
libgdk-x11-2.0.so.0.2400.32	0.71MB
libfreetype.so.6.15.0	0.7MB
libvorbisenc.so.2.0.11	0.66MB
libaspell.so.15.2.0	0.65MB
libpixman-1.so.0.34.0	0.64MB
libgegl-0.3.so.0.330.0	0.63MB
libgtksourceview-3.0.so.1.8.0	0.63MB
libharfbuzz.so.0.10702.0	0.62MB
libGLdispatch.so.0.0.0	0.58MB
libsystemd.so.0.21.0	0.51MB
libgmp.so.10.3.2	0.5MB
libpulsecommon-11.1.so	0.49MB
liborc-0.4.so.0.28.0	0.49MB
libzstd.so.1.3.3	0.48MB
libsndfile.so.1.0.28	0.46MB
libtiff.so.5.3.0	0.46MB
libFLAC.so.8.3.0	0.46MB
libgstbase-1.0.so.0.1405.0	0.46MB
libnl-route-3.so.200.24.0	0.45MB
Other libraries	27.56MB
Total	**197MB**

Table 9: 50 largest Low-level Libraries transformed by SBR