Compiler-assisted Code Randomization

Hyungjoon Koo,* Yaohui Chen,† Long Lu,† Vasileios P. Kemerlis,‡ Michalis Polychronakis*

*Stony Brook University
{hykoo, mikepo}@cs.stonybrook.edu

†Northeastern University
{yaohway, long}@ccs.neu.edu

‡Brown University
vpk@cs.brown.edu

***Abstract*—Despite decades of research on software diversification, only address space layout randomization has seen widespread adoption. Code randomization, an effective defense against return-oriented programming exploits, has remained an academic exercise mainly due to i) the lack of a transparent and streamlined deployment model that does not disrupt existing software distribution norms, and ii) the inherent incompatibility of program variants with error reporting, whitelisting, patching, and other operations that rely on code uniformity. In this work we present *compiler-assisted code randomization (CCR)*, a hybrid approach that relies on compiler–rewriter cooperation to enable fast and robust fine-grained code randomization on end-user systems, while maintaining compatibility with existing software distribution models. The main concept behind CCR is to augment binaries with a minimal set of transformation-assisting metadata, which i) facilitate rapid fine-grained code transformation at installation or load time, and ii) form the basis for reversing any applied code transformation when needed, to maintain compatibility with existing mechanisms that rely on referencing the original code. We have implemented a prototype of this approach by extending the LLVM compiler toolchain, and developing a simple binary rewriter that leverages the embedded metadata to generate randomized variants using basic block reordering. The results of our experimental evaluation demonstrate the feasibility and practicality of CCR, as on average it incurs a modest file size increase of 11.46% and a negligible runtime overhead of 0.28%, while it is compatible with link-time optimization and control flow integrity.**

I. Introduction

Diversifying the location and structure of code across different instances of a program is an effective approach for challenging the construction of reliable exploits that rely on return-oriented programming (ROP) [1]. Code diversification [2–11] breaks the attackers' assumptions about the ROP gadgets present in a vulnerable process, as even if the offsets of certain gadgets in the original program are known, the same offsets in a diversified variant of the same program will correspond to arbitrary instruction sequences, rendering ROP payloads constructed based on the original image unusable.

In applications with scripting capabilities, however, attackers can use script code that leverages a memory disclosure vulnerability to *on-the-fly* scan the code segments of the process, pinpoint useful gadgets, and synthesize them into a dynamically-constructed ROP payload customized for the given diversified variant. This "just-in-time" ROP (JIT-ROP) exploitation technique [12] can be used to bypass code diversification protections, or to just make ROP payloads more robust against frequent software updates, and is actively used in the wild [13–17]. Code diversification can also be bypassed under certain circumstances by remotely leaking [18–20] or inferring [21,22] what code exists at a given memory location. As a response, recent protections against JIT-ROP exploits rely on the concept of *execute-only* memory (XOM) [23,24] to allow instruction fetches but prevent memory reads from code pages, blocking this way the on-the-fly discovery of gadgets [25–33]. In fact, having an efficient way to enforce an "execute-no-read" policy on memory pages is a capability attractive enough that prompted hardware vendors to introduce hardware support in the form of an extra "execute" bit per memory page [34,35]. As an obvious—but important—observation, we note that any execute-only scheme is pointless if the protected code has not been *extensively* diversified.

Either as a standalone defense, or as a prerequisite of execute-only hardware and software memory protections, software diversification is indisputably an important and effective defense against modern exploits, as also evident by the vast body of work in this area [36]. Surprisingly, however, despite decades of research [37,38], only address space layout randomization (ASLR) [39,40] (and lately, link-time coarse-grained code permutation in OpenBSD [41]) has actually seen widespread adoption. More comprehensive techniques, such as fine-grained code randomization [4–10], have mostly remained academic exercises for two main reasons: i) lack of a transparent and streamlined deployment model for diversified binaries that does not disrupt existing software distribution norms; and ii) incompatibility with well-established software build, regression testing, debugging, crash reporting, diagnostics, and security monitoring workflows and mechanisms.

The vast majority of existing code diversification approaches rely on code recompilation [5, 32, 42, 43], static binary rewriting [7,8,10,33,44–46], or dynamic binary instrumentation [8,10,47,48] to generate randomized variants. The breadth of this spectrum of approaches stems from tradeoffs related to their applicability (source code is not always available), accuracy (code disassembly and control flow graph extraction are challenging for closed-source software) and performance (dynamic instrumentation incurs a high runtime overhead) [36]. From a deployment perspective, however, *all* above approaches share the same main drawbacks: i) the burden of diversification is placed on end users, who are responsible for carrying out a cumbersome process involving complex tools and operations, and requiring a significant amount of computational resources and human expertise; and ii) the diversified binaries become incompatible with patching, crash reporting, whitelisting, and other mechanisms that rely on software uniformity.

An alternative would be to let software vendors carry out the diversification process by delivering pre-randomized executables to end users. Under this model, the availability of source code makes even the most fine-grained forms of code randomization easily applicable, while the distribution of software variants could be facilitated through existing "app stores" [2, 3]. Although seemingly attractive due to its transparency, this deployment model is unlikely to be adopted in practice due to the increased cost that it imposes on vendors for generating and distributing program variants across millions or even billions of users [36, 49]. Besides the (orders-of-magnitude) higher computational cost for generating a new variant per user, a probably more challenging issue is that software mirrors, content delivery networks, and other caching mechanisms involved in software delivery become useless.

Aiming to combine the benefits of both approaches, in this work we propose *compiler-assisted code randomization (CCR)*, a hybrid method that involves the cooperation of both end users and software vendors for enabling the practical and transparent deployment of code randomization. At the vendor side, the compilation process of the "master binary" (which is meant to be released through existing distribution channels) is extended to collect and embed a set of *transformation-assisting metadata* into the final executable. At the client side, a rewriter component processes the augmented binary and rapidly generates a hardened variant by leveraging the embedded metadata, without performing any code disassembly or other complex binary analysis.

The possibility of a hybrid approach has been identified as a potentially attractive solution [36], and (to the best of our knowledge) we are the first to attempt an actual feasibility study of this concept by designing, implementing, and evaluating an end-to-end code transformation toolchain. We are not the first though to identify the benefits of augmenting binaries with transformation-assisting metadata, as previous works have proposed load-time diversification schemes based on self-randomizing binaries [5, 50]. The main drawbacks of these approaches are: i) the granularity of the performed randomization, which is limited to whole-function reordering; and ii) the lack of the backwards compatibility and flexibility that a separate rewriter component provides, which allows for selective hardening and fine-tuning according to the characteristics of each particular system.

Although an improvement over no code randomization at all, function permutation is not enough to prevent a wide range of attacks that rely on leaking code pointers to pinpoint gadgets within function bodies [19, 51–55]. It is indicative that recent execute-only memory defenses explicitly require much finer-grained randomization than function reordering for that exact purpose [20, 25, 27–33]. At the same time, function reordering is quite simple to perform, by just maintaining the existing information regarding function boundaries and code references that object files already contain, which would otherwise be discarded at link time. In contrast, as we show in this work, moving to more fine-grained randomization, such as basic block or instruction reordering, requires keeping a non-trivial amount of *additional* information that is stripped before the compiler generates each object file, let alone at link time.

Our work makes the following main contributions:

- We propose *compiler-assisted code randomization (CCR)*, a practical and generic code transformation approach that relies on compiler–rewriter cooperation to enable fast and robust diversification of binary executables on end-user systems.
- We have identified a minimal set of metadata that can be embedded into executables to facilitate rapid *fine-grained* code randomization at the basic block level, and maintain compatibility with existing mechanisms that rely on referencing the original code.
- We have designed and implemented an open-source prototype of CCR by extending the LLVM/Clang compiler and the GNU `gold` linker to generate augmented binaries, and developing a binary rewriter that leverages the embedded metadata to generate hardened variants. Our prototype supports existing features including (but not limited to) position independent code, shared objects, exception handling, inline assembly, lazy binding, link-time optimization, and even control flow integrity.
- We have experimentally evaluated our prototype and demonstrate its practicality, as on average it results in a modest file size increase of 11.46%, and incurs a negligible runtime overhead of 0.28%.

II. Motivation

Code randomization techniques can be categorized into two main types according to their deployment model—more specifically, according to *who* is responsible for randomizing the code (software vendor vs. end user) and *where* the actual randomization takes place (vendor's system vs. user's system). In this section, we discuss these two types of techniques in relation to the challenges that so far have prevented their deployment, and introduce our proposed approach.

A. Diversification by End Users

The vast majority of code randomization proposals shift the burden of diversification solely to end users, as *they* are responsible for diversifying the obtained software on *their* systems. For open-source software, this entails obtaining its source code, setting up a proper build environment, and recompiling the software with a special toolchain [4–6, 28, 32, 55, 56]. For closed-source software, this entails transforming existing executables using static binary rewriting, sometimes assisted by a runtime component to compensate for the imprecisions of binary code disassembly [7–9, 11, 20, 46, 47, 57]. Interested readers are referred to the survey of Larsen et al. [36] for an extensive discussion on the many challenges that code randomization techniques based on static binary rewriting face.

From a deployment perspective, however, both compiler-level and rewriter-level techniques share the same main drawbacks: end users (or system administrators) are responsible for diversifying an obtained application through a complex and oftentimes cumbersome process. In addition, this is a process that requires substantial computational and human resources in terms of the system on which the diversification will take place, as well as in terms of the time, effort, and expertise needed for configuring the necessary tools and performing the actual diversification. Consequently, it is unrealistic to expect this deployment model to reach the level of transparency that other diversification protections, like ASLR, have achieved.

At the same time, these approaches clash with operations that rely on software uniformity, which is an additional limiting factor against their deployment [3, 36]. When code randomization is applied at the client side, crash dumps and debug logs from randomized binaries refer to meaningless code and data addresses, code signing and integrity checks based on precomputed checksums fail, and patches and updates are not applicable on the diversified instances, necessitating the whole diversification process to be performed again from scratch.

B. *Diversification by Software Vendors*

Given that expecting end users to handle the diversification process is a rather unrealistic proposition for facilitating widespread deployment, an alternative is to rely on software vendors for handling the whole process and distributing already diversified binaries—existing app store software delivery platforms are particularly attractive for this purpose [2]. The great benefit of this model is that it achieves complete transparency from the perspective of end users, as they continue to obtain and install software as before [58]. Additionally, as vendors are in full control of the distribution process, they can alleviate any error reporting, code signing, and software update issues by keeping (or embedding) the necessary information for each unique variant to carry out these tasks [36].

Unfortunately, shifting the diversification burden to the side of software vendors also entails significant costs that in practice make this approach unattractive. The main reason is the increased cost for both *generating* and *distributing* diversified software variants [36]. Considering that popular software may exceed a billion users [49], the computational resources needed for generating a variant per user, per install, upon each new major release, can be prohibitively high from a cost perspective, even when diversification happens only at the late stages of compilation [3]. Additionally, as each variant is different, distribution channels that rely on caching, content delivery networks, software mirrors, or peer-to-peer transfers, will be rendered ineffective. Finally, at the release time of a new version of highly popular software, an issue of “enough inventory” will arise, as it will be challenging for a server-side diversification system to keep up with the increased demand in such a short time span [36].

C. *Compiler–Rewriter Cooperation*

The security community has identified compiler–rewriter cooperation as a potentially attractive solution for software diversification [36], but (to the best of our knowledge) no actual design and implementation attempt has been made before. We discuss in detail our design goals and the benefits of the proposed approach in Section IV-A.

Note that our aim is not to enable reliable code disassembly at the client side (which Larsen et al. [36] suggested as a possibility for a hybrid model), but to enable *rapid* and *safe* fine-grained code randomization by simply treating code as a sequence of raw bytes. In this sense, our proposal is more in line with the way ASLR has been deployed: developers must explicitly compile their software with ASLR support (i.e., with relocation information or using position-independent code), while the OS (if it supports ASLR) takes care of performing the actual transformation (i.e., the dynamic linker/loader maps each module to a randomly-chosen virtual address).

This flexibility and backwards compatibility is an important benefit compared to the alternative approach of self-randomizing binaries [5, 50]. According to the characteristics of each particular system, administrators may opt for randomization at installation or load time (or no randomization at all), and selectively enable or disable additional hardening transformations and instrumentation that may be available. On systems not equipped with the rewriter component, augmented binaries continue to work exactly as before.

III. Background

To fulfill our goal of generic, transparent, and fast fine-grained code randomization at the client side, there is a range of possible solutions that one may consider. In this section, we discuss why existing solutions are not adequate, and provide some details about the compiler toolchain we used.

A. *The Need for Additional Metadata*

Static binary rewriting techniques [7, 47, 57] face significant challenges due to indirect control flow transfers, jump tables, callbacks, and other code constructs that result in incomplete or inaccurate control flow graph extraction [59–61]. More generally applicable techniques, such as in-place code randomization [9, 11], can be performed even with partial disassembly coverage, but can only apply narrow-scoped code transformations, thereby leaving parts of the code non-randomized (e.g., complete basic block reordering is not possible). On the other hand, approaches that rely on dynamic binary rewriting to alleviate the inaccuracies of static binary rewriting [8, 10, 47, 48] suffer from increased runtime overhead.

A relaxation that could be made is to ensure programs are compiled with debug symbols and relocation information, which can be leveraged at the client side to perform code randomization. Symbolic information facilitates runtime debugging by providing details about the layout of objects, types, addresses, and lines of source code. On the other hand, it does not include *lower-level* information about complex code constructs, such as jump tables and callback routines, nor it contains metadata about (handwritten) assembly code [62]. To make matters worse, modern compilers attempt to generate cache-friendly code by inserting alignment and padding bytes between basic blocks, functions, objects, and even between jump tables and read-only data [63]. Various performance optimizations, such as profile-guided [64] and link-time [65] optimization, complicate code extraction even further—Bao et al. [66], Rui and Sekar [67], and others [68–70], have repeatedly demonstrated that accurately identifying functions (and their boundaries) in binary code is a challenging task.

In the same vein, Williams-King et al. [46] implemented Shuffler, a system that relies on symbolic and relocation information (provided by the compiler and linker) to disassemble code and identify all code pointers, with the goal of performing live code re-randomization. Despite the impressive engineering effort, its authors admit that they *"encountered myriad special cases"* related to inaccurate or missing metadata, special types of symbols and relocations, and jump table entries and invocations. Considering that these numerous special cases occurred just for a particular compiler (GCC), platform (x86-64 Linux), and set of (open-source) programs, it is reasonable to expect that similar issues will arise again, when moving to different platforms and more complex applications.

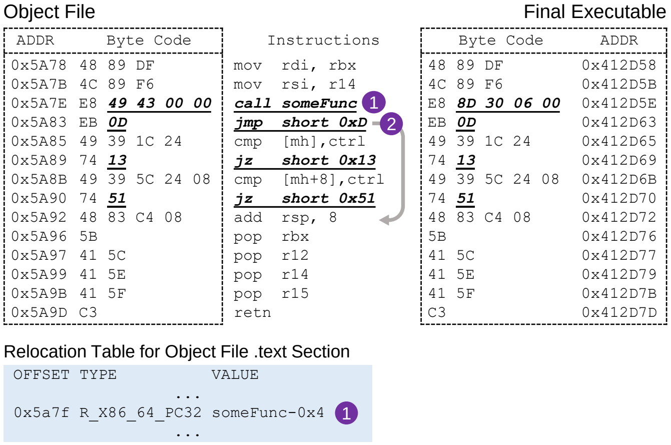
Based on the above, we argue that relying on *existing* compiler-provided metadata is not a viable approach for building a *generic* code transformation solution. More importantly, the complexity involved in the transformation process performed by the aforementioned schemes (e.g., static code disassembly, control flow graph extraction, runtime analysis, heuristics) is far from what could be considered reasonable for a *fast* and *robust* client-side rewriter, as discussed in Section II-A. Consequently, we opt for augmenting binaries with just the necessary *domain-specific metadata* needed to facilitate safe and generic client-side code transformation (and hardening) without any further binary code analysis.

B. Fixups and Relocations

When performing code randomization, machine instructions with register or immediate operands do not require any modification after they are moved to a new (random) location. In contrast, if an operand contains a (relative or absolute) reference to a memory location, then it has to be adjusted according to the instruction's new location, the target's new location, or both. (Note that a similar process takes place during the late stages of compilation.)

Focusing on LLVM, whenever a value that is not yet concrete (e.g., a memory location or an external symbol) is encountered during the instruction encoding phase, it is represented by a placeholder value, and a corresponding *fixup* is emitted. Each fixup contains information on how the placeholder value should be rewritten by the assembler when the relevant information becomes available. During the *relaxation* phase [71, 72], the assembler modifies the placeholder values according to their fixups, as they become known to it. Once relaxation completes, any unresolved fixups become *relocations*, stored in the resulting object file.

Figure 1 shows a code snippet that contains several fixups and one relocation. The left part corresponds to an object file after compilation, whereas the right one depicts the final executable after linking. Initially, there are four fixups (underlined bytes) emitted by the compiler. As the relocation table shows, however, only a single relocation (which corresponds to fixup ①) exists for address `0x5a7f`, because the other three fixups were resolved by the assembler. Henceforth, we explicitly refer to relocations in object files as *link-time relocations*—i.e., fixups that are left unresolved after the assembly process (to be handled by the linker). Similarly, we refer to relocations in executable files (or dynamic shared objects) as *load-time relocations*—i.e., relocations that are left unresolved after linking (to be handled by the dynamic linker/loader). Note that in this particular example, the final executable does not contain any load-time relocations, as relocation ① was resolved during linking (`0x4349`→`0x6308d`).



```
 ADDR      Byte Code           Instructions          Byte Code       ADDR
0x5A78 48 89 DF            mov  rdi, rbx            48 89 DF         0x412D58
0x5A7B 4C 89 F6            mov  rsi, r14            4C 89 F6         0x412D5B
0x5A7E E8 49 43 00 00      call someFunc            E8 8D 30 06 00   0x412D5E
0x5A83 EB 0D               jmp  short 0xD           EB 0D            0x412D63
0x5A85 49 39 1C 24         cmp  [mh],ctrl           49 39 1C 24      0x412D65
0x5A89 74 13               jz   short 0x13          74 13            0x412D69
0x5A8B 49 39 5C 24 08      cmp  [mh+8],ctrl         49 39 5C 24 08   0x412D6B
0x5A90 74 51               jz   short 0x51          74 51            0x412D70
0x5A92 48 83 C4 08         add  rsp, 8              48 83 C4 08      0x412D72
0x5A96 5B                  pop  rbx                 5B               0x412D76
0x5A97 41 5C               pop  r12                 41 5C            0x412D77
0x5A99 41 5E               pop  r14                 41 5E            0x412D79
0x5A9B 41 5F               pop  r15                 41 5F            0x412D7B
0x5A9D C3                  retn                     C3               0x412D7D
```

```
OFFSET TYPE          VALUE
                 ...
0x5a7f R_X86_64_PC32 someFunc-0x4
                 ...
```

Fig. 1. Example of the fixup and relocation information that is involved during the compilation and linking process.

In summary, load-time relocations are a subset of link-time relocations, which are a subset of all fixups. Unfortunately, even if link-time relocations are completely preserved by the linker, they are not sufficient for performing fine-grained code randomization. For instance, fixup ② is earlier resolved by the assembler, but is essential for basic block reordering, as the respective single-byte `jmp` instruction may have to be replaced by a four-byte one—if the target basic block is moved more than 127 bytes forward or 126 bytes backwards from the `jmp` instruction itself. Evidently, comprehensive fixups are *pivotal* pieces of information for fine-grained code shuffling, and should be promoted to first-class metadata by modern toolchains in order to provide support for generic, transparent, and compatible code diversification.

IV. Enabling Client-side Code Diversification

A. Overall Approach

Our design is driven by the following two main goals, which so far have been limiting factors for the actual deployment of code diversification in real-world environments:

Practicality: From a deployment perspective, a practical code diversification scheme should not disrupt existing features and software distribution models. Requiring software vendors to generate a diversified copy per user, or users to recompile applications from source code or transform them using complex binary analysis tools, have proven to be unattractive models for the deployment of code diversification.

Compatibility: Code randomization is a highly disruptive operation that should be safely applicable even for complex programs and code constructs. At the same time, code randomization inherently clashes with well-established operations that rely on software uniformity. These include security and quality monitoring mechanisms commonly found in enterprise settings (e.g., code integrity checking and whitelisting), as well as crash reporting, diagnostics, and self-updating mechanisms.

Augmenting compiled binaries with metadata that enable their subsequent randomization at installation or load time is an approach fully compatible with existing software distribution norms. The vast majority of software is distributed in the form of compiled binaries, which are carefully generated, tested, signed, and released through official channels by software vendors. On each endpoint, at installation time, the distributed software typically undergoes some post-processing and customization, e.g., its components are decompressed and installed in appropriate locations according to the system's configuration, and sometimes they are even further optimized according to the client's architecture, as is the case with Android's ahead-of-time compilation [73] or the Linux kernel's architecture-specific optimizations [74]. Under this model, code randomization can fittingly take place as an additional post-processing task during installation.

As an alternative, randomization can take place at load time, as part of the modifications that the loader makes to code and data sections for processing relocations [75]. However, to avoid extensive user-perceived delays due to the longer rewriting time required for code randomization, a more viable approach would be to maintain a supply of pre-randomized variants (e.g., an OS service can be generating them in the background), which can then instantly be picked by the loader.

Note that this distribution model is followed *even for open-source software*, as installing binary executables through package management systems (e.g., `apt-get`) offers unparalleled convenience compared to having to compile each new or updated version of a program from scratch. More importantly, under such a scheme, each endpoint can choose among different levels of diversification (hardening vs. performance), by taking into consideration the anticipated exposure to certain threats [76], and the security properties of the operating environment (e.g., private intranet vs. Internet-accessible setting).

The embedded metadata serves two main purposes. First, it allows the safe randomization of even complex software without relying on imprecise methods and incomplete symbolic or debug information. Second, it forms the basis for *reversing* any applied code transformation when needed, to maintain compatibility with existing mechanisms that rely on referencing the original code that was initially distributed.

Figure 2 presents a high-level view of the overall approach. The compilation process remains essentially the same, with just the addition of metadata collection and processing steps during the compilation of each object file and the linking of the final *master* executable. The executable can then be provided to users and endpoints through existing distribution channels and mechanisms, without requiring any changes.

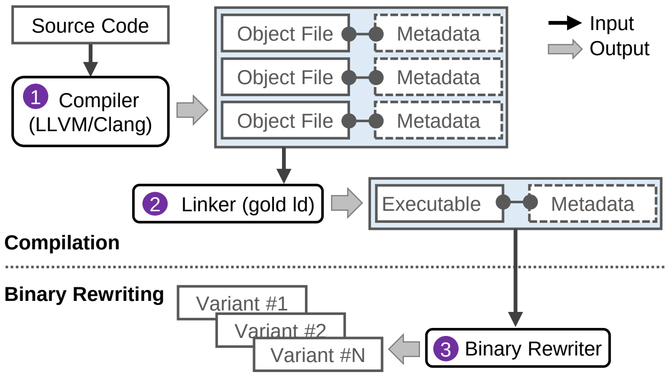


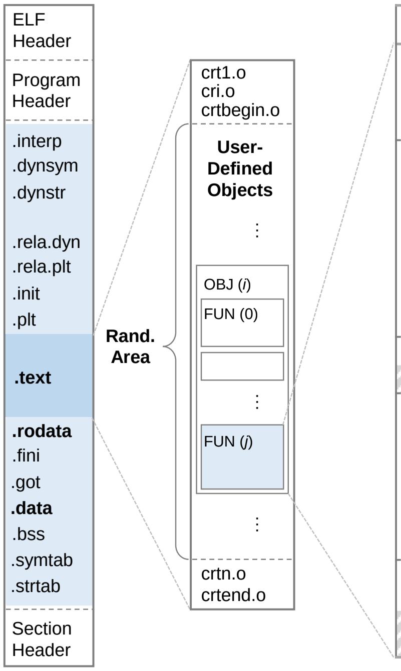
Fig. 2. Overview of the proposed approach. A modified compiler collects metadata for each object file ①, which is further updated and consolidated at link time into a single extra section in the final executable ②. At the client side, a binary rewriter leverages the embedded metadata to rapidly generate randomized variants of the executable ③.

As part of the installation process on each endpoint, a *binary rewriter* generates a randomized version of the executable by leveraging the embedded metadata. In contrast to existing code diversification techniques, this transformation does not involve any complex and potentially imprecise operations, such as code disassembly, symbolic information parsing, reconstruction of relocation information, introduction of pointer indirection, and so on. Instead, the rewriter performs simple transposition and replacement operations based on the provided metadata, treating all code sections as *raw binary data*. Our prototype implementation, discussed in detail in Section V, currently supports fine-grained randomization at the granularity of functions and basic blocks, is oblivious to any applied compiler optimizations, and supports static executables, shared objects, PIC, partial/full RELRO [77], exception handling, LTO, and even CFI.

B. Compiler-level Metadata

Our work is based on LLVM [78], which is widely used in both academia and industry, and we picked the ELF format and the x86-64 architecture as our initial target platform. Figure 3 illustrates an example of the ELF layout generated by Clang (LLVM's native C/C++/Objective-C compiler).

1) Layout Information: Initially, the range of the transformable area is identified, as shown in the left side of Figure 3. This area begins at the offset of the first object in the `.text` section and comprises all user-defined objects that can be shuffled. We modified LLVM to append a new section named `.rand` in every compiled object file so that the linker can be aware of which objects have embedded metadata. In our current prototype, we assume that all user-defined code is consecutive. Although it is possible to have intermixed code and data in the same section, we have ignored this case for now, as by default LLVM does not mix code and data when emitting `x86` code. This is the case for other modern compilers too—Andriesse et al. [79] could identify 100% of the instructions when disassembling GCC and Clang binaries (but CFG reconstruction still remains challenging).



BBL	Emitted Bytes by Clang (Fixup)	Disassembly by IDA Pro	Fragment
#0	0x40ABD0 53	push rbx	#0 (DF)
	0x40ABD1 48 8B 1D 58 F7 0B 00	mov rbx, cs:Fun1	
	0x40ABD8 48 85 DB	test rbx, rbx	
	0x40ABDB 74 ***2A***	jz short loc_40AC07	#1 (RF)
#1	0x40ABDD 48 89 DF	mov rdi, rbx ; s	#2 (DF)
	0x40ABE0 E8 7B D7 FF FF	call _strlen	
	0x40ABE5 48 89 DF	mov rdi, rbx ; b	
	0x40ABE8 48 89 C6	mov rsi, rax ; n	
	0x40ABEB E8 50 D3 00 00	call smemclr	
	0x40ABF0 48 8B 3D 39 F7 0B 00	mov rdi, cs:Fun1	
	0x40ABF7 E8 74 D3 00 00	call safefree	
	0x40ABFC 48 C7 05 29 F7 0B 00 00 00 00 00	mov cs:Fun1, 0	
#2	0x40AC07 31 DB	xor ebx, ebx	
	0x40AC09 0F 1F 80 00 00 00 00	nop dword ptr [rax+0x0h]	#3 (AF)
#3	0x40AC10 48 8B BB 40 A3 4C 00	mov rdi, qword ptr ds:Fun2[rbx]	#4 (DF)
	0x40AC17 E8 54 D3 00 00	call safefree	
	0x40AC1C 0F 57 C0	xorps xmm0, xmm0	
	0x40AC1F 0F 29 83 40 A3 4C 00	movaps xmmword ptr ds:Fun2[rbx], xmm0	
	0x40AC26 48 83 C3 10	add rbx, 10h	
	0x40AC2A 48 83 FB 20	cmp rbx, 20h	
	0x40AC2E 75 ***E0***	jnz short loc_40AC10	#5 (RF)
#4	0x40AC30 5B	pop rbx	#6 (DF)
	0x40AC31 C3	retn	
	0x40AC32 66 66 66 66 66 2E 0F 1F 84 00 00 00 00 00	align 20h	#7 (AF)

Fig. 3. An example of the ELF layout generated by Clang (left), with the code of a particular function expanded (center and right). The leftmost and rightmost columns in the code listing ("BBL" and "Fragment") illustrate the relationships between basic blocks and LLVM's various kinds of fragments: data (DF), relaxable (RF), and alignment (AF). Data fragments are emitted by default, and may span consecutive basic blocks (e.g., BBL #1 and #2). The relaxable fragment #1 is required for the branch instruction, as it may be expanded during the relaxation phase. The padding bytes at the bottom correspond to a separate fragment, although they do not belong to any basic block.

When loading a program, a sequence of startup routines assist in bootstrap operations, such as setting up environment variables and reaching the first user-defined function (e.g., `main()`). As shown in Figure 3, the linker appends several object files from `libc` into the executable for this purpose (`crt1.o`, `cri.o`, `crtbegin.o`). Additional object files include process termination operations (`crtn.o`, `crtend.o`). Currently, these automatically-inserted objects are out of transformation—this is an implementation issue that can be easily addressed by ensuring that a set of augmented versions of these objects is made available to the compiler. At program startup, the function `_start()` in `crt1.o` passes five parameters to `__libc_start_main()`, which in turn invokes the program's `main()` function. One of the parameters corresponds to a pointer to `main()`, which we need to adjust after `main()` has been displaced.

The metadata we have discussed so far are updated at link time, according to the final layout of all objects. The upper part of Table I summarizes the collected layout-related metadata.

2) Basic Block Information: The bulk of the collected metadata is related to the size and location of objects, functions, basic blocks (BBL), and fixups, as well as their relationships. For example, a fixup inherently belongs to a basic block, a basic block is a member of a function, and a function is included in an object. The LLVM backend goes through a very complex code generation process which involves all scheduled module and function passes for emitting globals, alignments, symbols, constant pools, jump tables, and so on. This process is performed according to an internal hierarchical structure of *machine functions*, *machine basic blocks*, and *machine instructions*. The machine code (MC) framework of the LLVM backend operates on these structures and converts machine instructions into the corresponding target-specific binary code. This involves the `EmitInstruction()` routine, which creates a new chunk of code at a time, called a *fragment*.

As a final step, the assembler (`MCAssembler`) assembles those fragments in a target-specific manner, decoupled from any logically hierarchical structure—that is, the unit of the assembly process is the fragment. We internally label each instruction with the corresponding parent basic block and function. The collection process continues until instruction relaxation has completed, to capture the emitted bytes that will be written into the final binary. As part of the final metadata, however, these labels are not essential, and can be discarded. As shown in Table I, we only keep information about the lower boundary of each basic block, which can be the end of an object (`OBJ`), the end of a function (`FUN`), or the beginning of the next basic block (`BBL`).

Going back to the example of Figure 3, we identify three types of *data*, *relaxable*, and *alignment* fragments, shown at the right side of the figure. The center of the figure shows the emitted bytes as generated by Clang, and their corresponding code as extracted by the IDA Pro disassembler, for the *j*-th function of the *i*-th object in the code section. The function consists of five basic blocks, eight fragments, and contains eleven fixups (underlined bytes).

TABLE I
COLLECTED RANDOMIZATON-ASSISTING METADATA

Metadata	Collected Information	Collection time
Layout	Section offset to first object Section offset to `main()` Total code size for randomization	Linking Linking Linking
Basic Block (BBL)	BBL size (in bytes) BBL boundary type (BBL, FUN, OBJ) Fall-through or not Section name that BBL belongs to	Linking Compilation Compilation Compilation
Fixup	Offset from section base Dereference size Absolute or relative Type (c2c, c2d, d2c, d2d) Section name that fixup belongs to	Linking Compilation Compilation Linking Compilation
Jump Table	Size of each jump table entry Number of jump table entries	Compilation Compilation

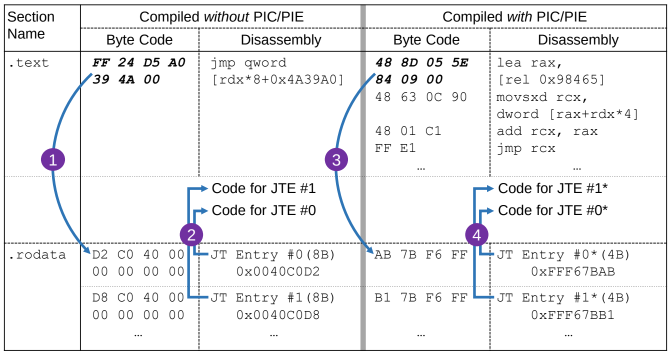
As discussed in Section III-B, relaxable fragments are generated only for branch instructions and contain just a single instruction. Alignment fragments correspond to padding bytes. In this example, there are two alignment fragments (#3 and #7): one between basic blocks #2 and #3, and one between function *j* and the following function. For metadata compactness, alignment fragments are recorded as part of the metadata for their preceding basic blocks. The rest of the instructions are emitted as part of data fragments.

Another consideration is *fall-through* basic blocks. A basic block terminated with a conditional branch implicitly falls through its successor depending on the evaluation of the condition. In Figure 3, the last instruction of BBL #0 jumps to BBL #2 when the zero flag is set, or control falls through to BBL #1. Such fall-through basic blocks must be marked so that they can be treated appropriately during reordering, as discussed in Section IV-D.

3) Fixup Information: Evaluating fixups and generating relocation entries are part of the last processing stage during layout finalization, right before emitting the actual code bytes. Note that this phase is orthogonal to the optimization level used, as it takes place after all LLVM optimizations and passes are done. Each fixup is represented by its offset from the section's base address, the size of the target (1, 2, 4, or 8 bytes), and whether it represents a relative or absolute value.

As shown in Table I, we categorize fixups into four groups, similar to the scheme proposed by Wang et al. [80], depending on their location (source) and the location of their target (destination): code-to-code (c2c), code-to-data (c2d), data-to-code (d2c), and data-to-data (d2d). We define data as a universal region that includes all other sections except the `.text` section. This classification helps in increasing the speed of binary rewriting when patching fixups after randomization, as discussed in Section IV-D.

4) Jump Table Information: Due to the complexity of some jump table code fragments, extra metadata needs to be kept for their correct handling during randomization. For non-PIC/PIE (position independent code/executable) binaries, the compiler generates jump table entries that point to targets using their absolute address. In such cases, it is trivial to update these destination addresses based on their corresponding fixups that already exist in the data section.



Section Name	Compiled *without* PIC/PIE		Compiled *with* PIC/PIE	
	Byte Code	Disassembly	Byte Code	Disassembly
.text	***FF 24 D5 A0*** ***39 4A 00***	jmp qword [rdx*8+0x4A39A0]	***48 8D 05 5E*** ***84 09 00*** 48 63 0C 90 48 01 C1 FF E1 ...	lea rax, [rel 0x98465] movsxd rcx, dword [rax+rdx*4] add rcx, rax jmp rcx ...
		Code for JTE #1 Code for JTE #0		Code for JTE #1* Code for JTE #0*
.rodata	D2 C0 40 00 00 00 00 00	JT Entry #0(8B) 0x0040C0D2	AB 7B F6 FF	JT Entry #0*(4B) 0xFFF67BAB
	D8 C0 40 00 00 00 00 00 ...	JT Entry #1(8B) 0x0040C0D8 ...	B1 7B F6 FF ...	JT Entry #1*(4B) 0xFFF67BB1 ...

Fig. 4. Example of jump table code generated for non-PIC and PIC binaries.

In PIC executables, however, jump table entries correspond to *relative* offsets, which remain the same irrespectively of the executable's load address. Figure 4 shows the code generated for a jump table when compiled without and with the PIC/PIE option. In the non-PIC case, the `jmp` instruction directly jumps to the target location ① by dereferencing the value of an 8-byte absolute address ② according to the index register `rdx`, as the address of the jump table is known at link time (`0x4A39A0`). On the other hand, the PIC-enabled code needs to compute the target with a series of arithmetic instructions. It first loads the base address of the jump table into `rax` ③, then reads from the table the target's relative offset and stores it in `rcx`, and finally computes the target's absolute address ④ by adding to the relative offset the table's base address.

To appropriately patch such jump table constructs, for which no additional information is emitted by the compiler, the only extra information we must keep is the number of entries in the table, and the size of each entry. This information is kept along with the rest of the fixup metadata, as shown in Table I, because the relative offsets in the jump table entries should be updated after randomization according to the new locations of the corresponding targets.

C. Link-time Metadata Consolidation

The main task of the linker is to merge multiple object files into a single executable. The linking process consists of three main tasks: constructing the final layout, resolving symbols, and updating relocation information. First, the linker maps the sections of each object into their corresponding locations in the final sections of the executable. During this process, alignments are adjusted and the size of extra padding for each section is decided. Then, the linker populates the symbol table with the final location of each symbol after the layout is finalized. Finally, it updates all relocations created by the assembler according to the final locations of those resolved symbols. These operations influence the final layout, and consequently affect the metadata that has already been collected at this point. It is thus crucial to update the metadata according to the final layout that is decided at link time.

Our CCR prototype is based on the GNU `gold` ELF linker that is part of `binutils`. It aims to achieve faster linking times compared to the GNU linker (`ld`), as it does not rely on the standard binary file descriptor (BFD) library. Additional advantages include lower memory requirements and parallel processing of multiple object files [81].

Figure 5 provides an overview of the linking process and the corresponding necessary updates to the collected metadata. Initially, the individual sections of each object are merged into a single one, according to the naming convention ①. For example, the two code sections `.text.obj1` and `.text.obj2` of the two object files are combined into a single `.text` section. Similarly, the metadata from each object is extracted and incorporated into a single section, and all addresses are updated according to the final layout ②.

As part of the section merging process, the linker introduces padding bytes between objects in the same section ③. At this point, the size of the basic block at the end of each object file has to be adjusted by increasing it according to the padding size. This is similar to the treatment of alignment bytes within an object file, which is considered as part of the preceding basic block (as discussed in Section IV-B2). Note that we do not need to update anything related to whole functions or objects, as our representation of the layout relies solely on basic blocks. Updating the size of the basic blocks that are adjacent to padding bytes is enough for deriving the final size of functions and objects.

Once the layout is finalized and symbols are resolved, the linker updates the relocations recorded by the assembler ④. Any fixups that were already resolved at compilation time are not available in this phase, and thus the corresponding metadata remains unchanged, while the rest is updated accordingly. Finally, the aggregation of metadata is completed ⑤ by updating the binary-level metadata discussed in Section IV-B, including the offset to the first object, the total code size for transformation, and the offset to the main function (if any).

A special case that must be considered is that a single object file may contain multiple `.text`, `.rodata`, `.data` or `.data.rel.ro` sections. For instance, C++ binaries often have several code and data sections according to a name mangling scheme, which enables the use of the same identifier in different namespaces. The compiler blindly constructs these sections without considering any possible redundancy, as it can only process the code of a single object file at a time. In turn, when the linker observes redundant sections, it nondeterministically keeps one of them and discards the rest [82]. This deduplication process can cause discrepancies in the layout and fixup information kept as part of our metadata, and thus the corresponding information about all removed sections is discarded at this stage. This process is facilitated by the section name information that is kept for basic blocks and fixups during compilation. Note that section names are optional attributes required only at link time. Consequently, after deduplication has completed, any remaining section name information about basic blocks and fixups is discarded, further reducing the size of the final metadata.

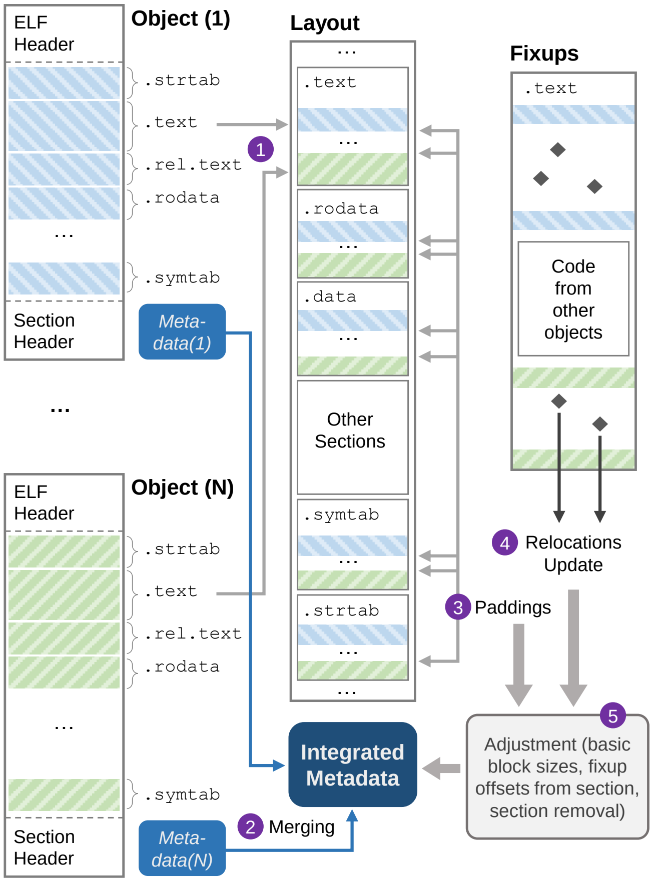


Fig. 5. Overview of the linking process. Per-object metadata is consolidated into a single section.

D. Code Randomization

To strike a balance between performance and randomization entropy, we have opted to maintain some of the constraints imposed by the code layout decided at link time, due to short fixup sizes and fall-through basic blocks. As mentioned earlier, these constraints can be relaxed by modifying the width of short branches and adding new branches when needed. However, our current choice has the simplicity and performance benefit of keeping the total size of code the same, which helps in maintaining caching characteristics due to spatial locality. To this end, we prioritize basic block reordering at intra-function level, and then proceed with function-level reordering.

Distance constraints due to fixup size may occur in both function and basic block reordering. For instance, it is typical for functions to contain a short fixup that refers to a *different* function, as part of a jump instruction used for tail-call optimization. At the rewriting phase, basic block reordering proceeds without any constraints if: (a) the parent function of a basic block does not have any distance-limiting fixup, or (b) the size of the function allows reaching all targets of any contained short fixups. Note that the case of multiple functions sharing basic blocks, which is a common compiler optimization, is fully supported.

From an implementation perspective, the simplest solution for fall-through basic blocks is to assume that both child blocks will be displaced away, in which case an *additional* jump instruction must be inserted for the previously fall-through block. From a performance perspective, however, a better solution is to avoid adding any extra instructions and keep *either* of the two child basic blocks adjacent to its parent—this can be safely done by inverting the condition of the branch when needed. In our current implementation we have opted for this second approach, but have left branch inversion as part of our future work. As shown in Section VI-E, this decision does not impact the achieved randomization entropy.

After the new layout is available, it is essential to ensure fixups are updated accordingly. As discussed in Section IV-B3, we have classified fixups into four categories: c2c, c2d, d2c and d2d. In case of d2d fixups, no update is needed because we diversify only the code region, but we still include them as part of the metadata in case they are needed in the future. The dynamic linking process relies on c2d (relative) fixups to adjust pointers to shared libraries at runtime.

V. Implementation

Our CCR prototype supports ELF executables for the Linux x86-64 platform. To augment binaries, we modified LLVM/Clang v3.9.0 [78] and the `gold` linker v2.27 of GNU Binutils [83]. At the user side, binary executable randomization is performed by a custom binary rewriter that leverages the embedded metadata. In this section, we discuss the main modifications that were required in the compiler and linker, and the design of our binary rewriter. We encountered many challenges and pitfalls in our attempt to maintain compatibility with advanced features such as inline assembly, lazy binding, exception handling, link-time optimization, and additional protections like control flow integrity. Interested readers can find further details regarding those issues in the Appendix.

a) Compiler: In our attempt to modify the right spots in LLVM for collecting the necessary metadata, we encountered several challenges. First, as explained in Section IV-B2, the assembler operates on an entirely separate view based on fragments and sections, compared to the logical view of basic blocks and functions. For this reason, we had to modify the LLVM backend itself, rather than writing an LLVM pass, which would be more convenient, as LLVM offers a flexible interface for implementing optimizations and transformations.

Second, recall that fine-grained randomization necessitates absolute accuracy when it comes to basic block sizes. A single misattributed byte can result in the whole code layout being incorrect. In this regard, obtaining the exact size of each instruction is important for deriving the right sizes of both its parent basic block and function. In our implementation, extracting this information relies on labeling the parents of each and every instruction. However, we encountered several cases of instructions not belonging to any basic block. For example, sequences like `cld; rep stos;` may appear without any parent label. These are handled by including the instructions as part of the basic block of the previous instruction.

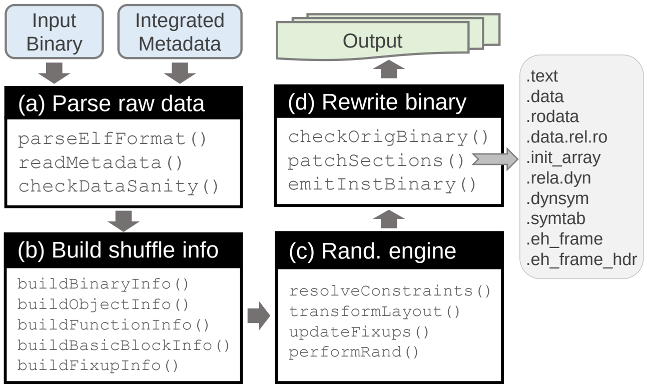


Fig. 6. Overview of the rewriting process. The rewriter parses the augmented ELF binary (a) and organizes all information required for randomization in a tree data structure (b). Randomization is performed based on this structure (c), and the new layout is then written into the final binary (d).

b) Linker: The linker performs several operations that influence considerably the final binary layout, and many of them required special consideration. First, there are cases of object files with zero size, e.g., when a source code file contains just a definition of a structure, without any actual code. Interestingly, such objects result in padding bytes that must be carefully accounted for when randomizing the last basic block of an object. Besides removing the metadata for redundant sections due to the deduplication process (discussed in Section IV-C), there are other sections that require special handling. These include `.text.unlikely`, `.text.exit`, `.text.startup`, and `.text.hot`, which the GNU linker handles differently for compatibility purposes. The special sections have unique features including independent positions (ahead of all other code) and redundant section names within a single object file (i.e., multiple `.text.startup` sections), resulting in non-consecutive user-defined code in the `.text` section that must be precisely captured as part of our metadata for randomization to function properly.

c) Binary Rewriter: We developed our custom binary rewriter in Python, and used the `pyelftools` library for parsing ELF files [84]. The rewriter takes the augmented ELF executable to be randomized as its sole input. The core randomization engine is written in ∼2KLOC. The simple nature of the rewriter makes it easy to be integrated as part of existing software installation workflows. In our prototype, we have integrated it with Linux's `apt` package management system through `apt`'s wrapper script functionality.

As illustrated in Figure 6, binary rewriting comprises four phases. Initially, the ELF binary is parsed and some sanity checks are performed on the extracted metadata. We employ Protocol Buffers for metadata serialization, as they provide a clean, efficient, and portable interface for structured data streams [85]. To minimize the overall size of the metadata, we use a compact representation by keeping only the minimum amount of information required. For example, as discussed in Section IV-B2, basic block records denote whether they belong to the end of a function or the end of an object (or both),

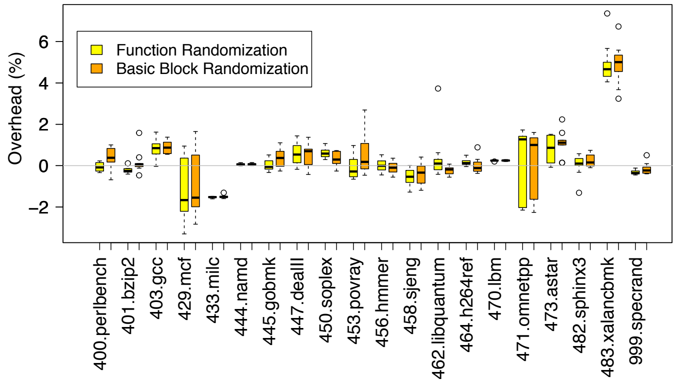


Fig. 7. Performance overhead of fine-grained (function vs. basic block reordering) randomization for the SPEC CPU2006 benchmark tests.

without keeping any extra function or object information per block. The final metadata is stored in a special `.rand` section, which is further compressed using `zlib`. Next, all information regarding the relationships between objects, functions, basic blocks, and fixups is organized in an optimized data structure, which the randomization engine uses to transform the layout, resolve any constraints, and update target locations.

VI. Experimental Evaluation

We evaluated our CCR prototype in terms of runtime overhead, file size increase, randomization entropy, and other characteristics. Our experiments were performed on a system equipped with an Intel i7-7700 3.6GHz CPU, 32GB RAM, running the 64-bit version of Ubuntu 16.04.

A. Randomization Overhead

We started by compiling the entire SPEC CPU2006 benchmark suite (20 C and C++ programs) with our modified LLVM and `gold` linker, using the `-O2` optimization level and without the PIC option. Next, we generated 20 different variants of each program, 10 using function reordering and 10 more using function and basic block reordering. Each run was performed 10 times for the original programs, and a single time for each of the 20 variants.

Figure 7 shows a boxplot of the runtime overhead for function reordering and basic block reordering. The dark horizontal line in each box corresponds to the median overhead value, which mostly ranges between zero and one across all programs. The top and bottom of each box correspond to the upper and lower quartile, while the whiskers to the highest and lowest value, excluding outliers, which are denoted by small circles (there were 14 such cases out of the total 400 variants, exhibiting an up to 7% overhead). Overall, the average performance overhead is negligible at 0.28%, with a 1.37 standard deviation. The average overhead per benchmark is reported in Table II, which also includes further information about the layout and fixups of each program.

Interesting cases are `mcf` and `milc`, the variants of which consistently exhibit a slight performance improvement, presumably due to better cache locality (we performed an extra round of experiments to verify it). In contrast, `xalancbmk` exhibited a distinguishably high average overhead of 4.9%. Upon further investigation, we observed a significant increase in the number of L1 instruction cache misses for its randomized instances. Given that `xalancbmk` is one of the most complex benchmarks, with a large number of functions and heavy use of indirect control transfers, it seems that the disruption of cache locality due to randomization has a much more pronounced effect. For such cases, it may be worth exploring profile-guided randomization approaches that will preserve the code locality characteristics of the application.

B. ELF File Size Increase

Augmenting binaries with additional metadata entails the risk of increasing their size at levels that may become problematic. As discussed earlier, this was an issue that we took into consideration when deciding what information to keep, and optimized the final metadata to include only the minimum amount of information necessary for code diversification.

As shown in Table II, file size increase ranges from 1.68% to 20.86%, with an average of 11.46% (13.3% for the SPEC benchmarks only). We consider this a rather modest increase, and do not expect it to have any substantial impact to existing software distribution workflows. The Layout columns (Objs, Funcs, BBLs) show the number of object files, functions, and basic blocks in each program. As expected, the metadata size is proportional to the size of the original code. Note that the generated randomized variants do not include any of the metadata, so their size is the same as the original binary.

C. Binary Rewriting Time

We measured the rewriting time of our CCR prototype by generating 100 variants of each program and reporting the average processing time. We repeated the experiment twice, using function and basic block reordering, respectively. As shown in Table II (Rewriting columns) the rewriting process is very quick for small binaries, and the processing time increases linearly with the size of the binary. The longest processing time was observed for `xalancbmk`, which is the largest and most complex (in terms of number of basic blocks and fixups) among the tested binaries. All but four programs were randomized in under 9s, and more than half of them in under 1s.

The reported numbers include the process of updating the debug symbols present in the `.symtab` section. As this is not needed for production (stripped) binaries, the rewriting time in practice will be shorter—indicatively, for `xalancbmk`, it is 30% faster when compiled without symbols. Note that our rewriter is just a proof of concept, and further optimizations are possible. Currently, the rewriting process involves parsing the raw metadata, building it into a tree representation, resolving any constraints in the randomized layout, and generating the final binary. We believe that the rewriting speed can be further optimized by improving the logic of our rewriter's randomization engine. Moving from Python to C/C++ is also expected to increase speed even further.

D. Correctness

To ensure that our code transformations do not affect in any way the correctness of the resulting executable, in addition to the SPEC benchmarks, we compiled and tested the augmented versions of ten real-world applications. For example, we parsed the entire LLVM source code tree with a randomized version of `ctags` using the `-R` (recursive) option. The MD5 hash of the resulting index file, which was 54MB in size, was identical to the one generated using the original executable. Another experiment involved the command-line audio encoding tool `oggenc`—a large and quite complex program (58,413 lines of code) written in C [86]—to convert a 44MB WAV file to the OGG format, which we then verified that was correctly processed. Furthermore, we successfully compiled popular server applications (web, FTP, and SSH daemons), confirming that their variants did not malfunction when using their default configurations. Application versions and the exact type of activity used for functionality testing are provided in Table III in the Appendix.

E. Randomization Entropy

We briefly explore the randomization entropy that can be achieved using function and basic block reordering, when considering the current constraints of our implementation. Let F_{ij} be the jth function in the ith object, f_i the number of functions in that object, and b_{ij} the number of basic blocks in the function F_{ij}. Suppose there are p object files comprising a given binary executable. The total number of functions q and basic blocks r in the binary can be written as $q = \sum_{i=0}^{p-1} f_i$ and $r = \sum_{i=0}^{p-1} \sum_{j=0}^{f_i-1} b_{ij}$. Then, the number of possible variants with function reordering is $q!$ and with basic block reordering is $r!$. Due to the large number of variants, let the randomization entropy E be the base 10 logarithm of the number of variants. In our case, we perform basic block randomization at intra-function level first, followed by function reordering. Therefore, the entropy can be computed as follows:

$$E = \log_{10}(\prod_{i=0}^{p-1}(\prod_{j=0}^{f_i-1} b_{ij}!) \cdot (\sum_{i=0}^{p-1} f_i)!)$$

However, as discussed in Section IV-D, our current implementation has some constraints regarding the placement of functions and basic blocks. Let the number of such function constraints in the ith object be y_i. Likewise, fall-through blocks are currently displaced together with their previous block. Similarly to functions, in some cases the size of a fixup also constrains the maximum distance to the referred basic block. Let the number of such basic block constraints in function F_{ij} be x_{ij}. Given the above, the entropy in our case can be calculated as:

$$E = \log_{10}(\prod_{i=0}^{p-1}(\prod_{j=0}^{f_i-1} (b_{ij} - x_{ij})!) \cdot (\sum_{i=0}^{p-1} (f_i - y_i))!)$$

Using the above formula, we report the randomization entropy for function and basic block level randomization in Table II. We observe that even for small executables like `lbm`, the number of variants exceeds 300 trillion. Consequently, our current prototype achieves more than enough entropy, which can be further improved by relaxing the above constraints (e.g., by separating fall-through basic blocks from their parent blocks, and adding a relaxation-like phase in the rewriter to alleviate existing fixup size constraints).

VII. Limitations

Our prototype implementation demonstrates the feasibility of CCR by enabling practical fine-grained code randomization (basic block reordering) on a popular platform (x86-64 Linux). There are, of course, several limitations, which can be addressed with additional engineering effort and are part of our future work.

First, individual assembly source code files (`.s`) are currently not supported. Note that assembly code files differ from inline assembly (which is fully supported), in that their processing by LLVM is not part of the standard abstract syntax tree and intermediate representation workflow, and thus corresponding function and basic block boundaries are missing during compilation. Still, symbols for functions contained in `.s` files are available, and we plan to include this information as part of the collected metadata.

Second, any use of self-modifying code is not supported, as the self-modification logic should be changed to account for the applied randomization. In such cases, compatibility can still be maintained by excluding (i.e., "pinning" down) certain code sections or object files from randomization, assuming all their external dependencies are included.

A slightly more important issue is fully updating all symbols contained in the debug sections according to the new layout after rewriting. Our current CCR prototype does update symbol table entries contained in the `.symtab` section, but it does not fully support the ones in the `.debug_*` sections. Although in practice the lack of full debug symbols is not a problem, as these are typically stripped off production binaries, this is certainly a useful feature to have. In fact, we were prompted to start working on resolving this issue because the lack of correct debug symbols for newly generated variants hindered our debugging efforts during the development of our prototype.

Finally, our prototype does not support programs with custom exception handling when randomization at the basic block level is used (this is not an issue for function-level randomization). No additional metadata is required to support this feature (just additional engineering effort). Further details about exception handling are provided in the Appendix.

VIII. Discussion

a) Other types of code hardening: Basic block reordering is an impactful code randomization technique that ensures that no ROP gadgets remain in their original locations, even *relatively* to the entry point of the function that includes them—an important aspect for defending against (indirect) JIT-ROP attacks that rely on code pointer leakage [19, 20, 53]. For a function that consists of just a single basic block, however, the relative distance of any gadgets from its entry

TABLE II
EXPERIMENTAL EVALUATION DATASET AND RESULTS (* INDICATES PROGRAMS WRITTEN IN C++)

Program	Layout			Fixups				Size (KB)			Rewriting (sec)		Overhead		Entropy ($\log_{10}$)	
	Objs	Funcs	BBLs	.text	.rodata	.data	.init_ar.	Orig.	Augm.	Increase	Func	BBL	Func	BBL	Func	BBL
400.perlbench	50	1,660	46,732	70,653	7,872	1,765	0	1,198	1,447	20.86%	7.69	8.05	-0.07%	0.32%	4,530	5,011
401.bzip2	7	71	2,407	2,421	75	0	0	90	101	12.80%	0.19	0.21	-0.23%	0.16%	100	157
403.gcc	143	4,326	118,397	189,543	84,357	367	0	3,735	4,465	19.54%	52.30	53.89	0.82%	0.91%	13,657	16,483
429.mcf	11	24	375	410	0	0	0	22	25	12.02%	0.08	0.09	-1.27%	-0.98%	23	44
433.milc	68	235	2,613	5,980	50	36	0	148	170	14.94%	0.48	0.50	-1.53%	-1.50%	456	600
444.namd*	23	95	7,480	8,170	24	0	0	312	345	10.49%	0.50	0.56	0.06%	0.07%	148	187
445.gobmk	62	2,476	25,069	44,136	1,377	21,400	0	3,949	4,116	4.23%	21.28	20.43	0.05%	0.35%	7,272	8,271
447.dealII*	6,295	6,788	100,185	103,641	7,954	1	45	4,217	4,581	8.65%	38.08	39.18	0.60%	0.52%	23,064	25,601
450.soplex*	299	889	13,741	15,586	1,561	0	61	467	531	13.76%	1.90	1.99	0.60%	0.28%	2,234	2,983
453.povray*	110	1,537	28,378	47,694	10,398	617	1	1,223	1,406	14.92%	5.67	5.88	-0.08%	0.50%	4,130	4,939
456.hmmer	56	470	10,247	14,265	798	156	0	343	400	16.53%	1.14	1.19	0.00%	-0.11%	1,042	1,313
458.sjeng	119	132	4,469	8,978	431	0	0	155	186	19.93%	0.50	0.53	-0.55%	-0.38%	221	334
462.libquantum	16	95	1,023	1,373	319	0	0	55	62	13.57%	0.19	0.19	0.40%	-0.24%	148	207
464.h264ref	42	518	14,476	23,180	320	321	0	698	782	12.01%	1.97	2.06	0.17%	0.00%	1,180	1,468
470.lbm	2	17	133	227	0	0	0	22	24	8.15%	0.06	0.06	0.25%	0.25%	14	24
471.omnetpp*	366	1,963	22,118	34,212	3,411	240	75	843	952	12.95%	4.73	4.94	0.03%	0.25%	5,560	6,983
473.astar*	14	88	1,116	1,369	6	1	0	56	62	12.03%	0.17	0.17	0.78%	1.08%	134	169
482.sphinx3	44	318	5,557	9,046	26	207	0	213	249	16.54%	0.68	0.72	0.02%	0.23%	656	815
483.xalancbmk*	3,710	13,295	130,691	142,128	19,936	323	0	6,217	6,836	9.95%	88.09	89.94	4.92%	4.89%	48,863	61,045
999.specrand	2	3	11	32	0	0	0	8	9	11.07%	0.03	0.03	-0.32%	-0.15%	0.8	1.6
ctags	50	423	8,550	13,618	3,733	507	0	795	851	7.03%	1.17	1.21	-	-	915	1,095
gzip	34	103	2,895	5,466	466	21	0	267	289	8.13%	0.40	0.41	-	-	164	194
lighttpd	50	351	5,817	9,169	818	98	0	866	903	4.23%	0.96	0.99	-	-	732	891
miniweb	7	67	1,322	1,681	65	74	0	56	64	14.54%	0.19	0.19	-	-	94	113
oggenc	1	428	7,035	7,746	183	3,869	0	2,120	2,156	1.68%	2.79	2.74	-	-	942	2,285
openssh	122	1,135	18,262	29,815	2,442	90	0	2,144	2,248	4.83%	4.04	4.17	-	-	3,398	3,856
putty	79	1,288	20,796	31,423	3,126	118	0	1,069	1,184	10.78%	3.71	3.82	-	-	2,927	3,610
vsftpd	39	516	3,793	7,148	74	0	0	138	163	18.48%	0.65	0.67	-	-	1,147	1,227
libcapstone	42	402	21,454	47,299	13,002	5	0	2,777	2,931	5.69%	10.64	11.31	-	-	863	1,040
dosbox*	630	3,127	66,522	124,814	14,906	2,585	18	11,729	12,145	3.54%	37.59	38.12	-	-	9,503	10,941

point still remains the same. This issue can be trivially addressed by modifying our rewriter to insert a (varying) number of NOPs or junk instructions at the beginning of the function [32]. Other more narrow-scope transformations, such as instruction substitution, intra basic block instruction reordering, and register reassignment [9, 36] can also be supported effortlessly, since our metadata provides precise knowledge about the boundaries of basic blocks. In fact, we have started leveraging such metadata for augmenting our rewriter with *agile* hardening capabilities: that is, strip (or not) hardening instrumentation (e.g., CFI [87], XOM [32]) based on where the target application is going to be deployed, thereby enabling precise and targeted protection.

Defending against more sophisticated attacks that rely on whole-function reuse [53, 88–92] requires more aggressive transformations, such as code pointer indirection [28,55,93,94] or function argument randomization. We leave the exploration of how our metadata could be extended to facilitate such advanced protections as part of future research.

b) Error reporting, whitelisting, and patching: One of the main benefits of code randomization based on compiler–rewriter cooperation is that it allows for maintaining compatibility with operations that rely on software uniformity, which currently is a major roadblock for its practical deployment. By performing the actual diversification on endpoints, any side-effects that hinder existing norms can be reversed.

For instance, a crash dump of a diversified process can be post-processed right after it is generated so that code addresses are changed to refer to the original code locations of the master binary that was initially distributed (otherwise, it will be of no use to its developers). Similarly, code integrity checking and whitelisting mechanisms can be modified to de-randomize the in-memory or on-disk code before actually verifying it. This randomization reversal process can be supported by including a randomization seed within each variant (which in conjunction with the original metadata will provide all the necessary information for the task) [36]. The seed can be kept as part of the on-disk binary (i.e., it does not need to exist in memory), to prevent attackers from getting any extra information about the randomized layout, e.g., through a memory disclosure vulnerability.

Code *signing* does not require any modification, since master binaries can continue to be signed normally before distribution. At the client side, the binary rewriter can proceed only after verifying the signature. Binary-level software patching is also not significantly affected. Patches can continue to be released in the same way as before, based on the master binary. At the client side, the patch can be applied on the master binary, and then a new (updated) variant can be generated.

c) Intellectual property: As an outcome of the compilation process, most of the high-level programming language structure and semantics are lost from the resulting binary code. Especially for proprietary software, the inherent complexity of code disassembly combined with the lack of symbolic information (and the potential use of code obfuscation) can hinder significantly any attempts of reconstructing the original code semantics through binary code disassembly, control flow graph extraction, and decompilation.

The metadata needed to facilitate code randomization can certainly aid in extracting a more accurate view of the assembly code and the control flow graph of a binary, but does not convey any new symbolic information that would help in extracting higher-level program semantics (function and variable names aid reverse engineering significantly). We do not consider this issue a major concern, as vendors who care about protecting their intellectual property against reverse engineering rely on more aggressive code obfuscation techniques (e.g., software packing or instruction virtualization). Alternatively, parts of code or whole modules for which such concerns apply can be excluded so that no additional metadata is kept for them.

IX. Related Work

Software diversity has been studied for decades in the context of security and reliability. Early works on software diversification focused on building fault-tolerant software for reliability purposes [95,96]. Changing the location of code can also improve performance, especially when guided by dynamic profiling [43,97–99]. In the security field, software diversification has received attention as a means of breaking software monocultures and mitigating mass exploitation [36–38].

Client-side code randomization often involves complex binary code analysis, which faces significant challenges when it comes to accuracy and coverage, especially when supplemental information (e.g., relocation or symbolic information) is not available [66, 68, 79, 100–102]. Static binary rewriting of stripped binaries is still possible in certain cases, although it involves either code extraction heuristics [7–11,33,44,47], or dynamic binary instrumentation [8,10,47,48]. Other implementation approaches include compile-time [5,28,32,42,43], link-time [6], load-time [7,8,10,33,44,50], and run-time [45,46, 103–105] solutions. On the other hand, the concept of server-side diversification has been briefly explored, especially as part of "app store" software distribution models [2,3].

From the above, probably the closest in spirit to our work are the technique proposed by Bhatkar et al. [5] and Selfrando [50], which both rely on a single compilation to generate self-randomizing binaries. Selfrando, for instance, uses a linker wrapper script to extract function boundary information from object files, which is maintained in the resulting executable. As mentioned earlier, these approaches are limited to function-level permutation, which is not enough for thwarting exploits that rely on code pointer leakage to infer the location of gadgets within functions [51–55]. As we have shown, basic block reordering is a much more complex process that requires additional metadata that must be extracted at earlier and later stages of the compilation process. Compared to function-level reordering, our approach achieves orders of magnitude higher randomization entropy, while the space overhead due to the metadata is actually lower (e.g., just 13.3% on average for the SPEC2006 benchmarks, compared to 24.7% for Selfrando). We should also note that due to their fundamental design decisions, i.e., the extraction of existing information from object files *after* they are compiled, these approaches cannot support link-time optimization, and consequently, any additional features that rely on it, such as CFI.

Binary rewriting schemes like XIFER [10] suffer from the problem of imprecision, given that 100% accurate disassembly and CFG extraction for complex C/C++ binaries (even when relocation information is available) is not possible. This is evident by the results of Andriesse et al. [70, 79], or the multitude of heuristics employed by Shuffler [46].

The recent renewed focus on code diversification led to the emergence of JIT-ROP attacks [12], which in turn led to the development of execute-only memory protections [25–33]. A prerequisite for these techniques is that the protected code must have been previously diversified using *fine-grained* randomization, which motivates our work. Although execute-only memory prevents code discovery, adversaries can still harvest code pointers from (readable) data sections and indirectly infer the location of code fragments [19,20,53], or in some implementations [29,30], achieve the same by partially reading or reloading pieces of code [22,106]. As a response, leakage-resilient diversification [28,55] combines execute-only memory with code pointer hiding through additional control flow indirection. Still, although the introduced indirection prevents harvested pointers from revealing anything useful about the immediate surrounding code area of their target, attackers may still be able to reuse whole functions, e.g., using harvested pointers to other functions of the same or lower arity [88,89,92].

X. Conclusion

Aiming to combine the benefits of compiler-level and binary-level code randomization techniques, we have presented a hybrid approach that relies on compiler–rewriter cooperation to facilitate fast and robust code diversification at the client side, by augmenting binaries with transformation-assisting metadata. We hope our work will alleviate the concerns of the broader security community regarding to the hurdles that until now have prevented the actual deployment of protections based on client-side fine-grained code transformation. We anticipate (and are currently working on) a first real-world deployment by integrating our binary rewriter into the `apt` package manager through its installation script support. To help the community reach this goal, we make our CCR prototype publicly available.

Availability

Our prototype open-source implementation is available at: https://github.com/kevinkoo001/CCR.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable feedback, and Christopher Morales for integrating CCR with `apt`. This work was supported by the Office of Naval Research (ONR) under awards N00014-15-1-2378, N00014-17-1-2788, N00014-17-1-2891, and N00014-18-1-2043. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the US government or ONR.

REFERENCES

[1] 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 (CCS)*, 2007, pp. 552–561.

[2] M. Franz, "E unibus pluram: Massive-scale software diversity as a defense mechanism," in *Proceedings of the New Security Paradigms Workshop (NSPW)*, 2010, pp. 7–16.

[3] P. Larsen, S. Brunthaler, and M. Franz, "Security through diversity: Are we there yet?" *IEEE Security Privacy*, vol. 12, no. 2, pp. 28–35, Mar 2014.

[4] E. Bhatkar, D. C. Duvarney, and R. Sekar, "Address obfuscation: an efficient approach to combat a broad range of memory error exploits," in *In Proceedings of the 12th USENIX Security Symposium*, 2003, pp. 105–120.

[5] S. Bhatkar, R. Sekar, and D. C. DuVarney, "Efficient techniques for comprehensive protection from memory error exploits," in *Proceedings of the 14th USENIX Security Symposium*, August 2005, pp. 255–270.

[6] 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 (ACSAC)*, 2006, pp. 339–348.

[7] R. Wartell, V. Mohan, K. W. Hamlen, Z. Lin, and W. C. Rd, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *Proceedings of the 19th ACM conference on Computer and communications security (CCS)*, 2012, pp. 157–168.

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

[9] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *Proceedings of the 33rd IEEE Symposium on Security & Privacy (S&P)*, May 2012, pp. 601–615.

[10] L. V. Davi, A. Dmitrienko, S. Nürnberger, and A.-R. Sadeghi, "Gadge me if you can: Secure and efficient ad-hoc instruction-level randomization for x86 and arm," in *Proceedings of the 8th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2013, pp. 299–310.

[11] H. Koo and M. Polychronakis, "Juggling the gadgets: Binary-level code randomization using instruction displacement," in *Proceedings of the 11th ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2016, pp. 23–34.

[12] K. Z. Snow, L. Davi, A. Dmitrienko, C. Liebchen, F. Monrose, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *Proceedings of the 34th IEEE Symposium on Security & Privacy (S&P)*, 2013, pp. 574–588.

[13] C. Pierce, "Another 0day, another prevention," https://www.endgame.com/blog/another-0day-another-prevention, 2016.

[14] A. Fobian and C.-B. Bender, "Firefox 0-day targeting Tor-users," https://blog.gdatasoftware.com/2016/11/29346-firefox-0-day-targeting-tor-users, 2016.

[15] M. Labs, "MWR Labs Pwn2Own 2013 Write-up - Webkit Exploit," 2013, https://labs.mwrinfosecurity.com/blog/mwr-labs-pwn2own-2013-write-up-webkit-exploit/.

[16] V. Kotov, "Dissecting the newest IE10 0-day exploit (CVE-2014-0322)," Feb. 2014, http://labs.bromium.com/2014/02/25/dissecting-the-newest-ie10-0-day-exploit-cve-2014-0322/.

[17] B. Antoniewicz, "Analysis of a Malware ROP Chain," Oct. 2013, http://blog.opensecurityresearch.com/2013/10/analysis-of-malware-rop-chain.html.

[18] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking blind," in *Proceedings of the 35th IEEE Symposium on Security & Privacy (S&P)*, 2014, pp. 227–242.

[19] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, M. Negro, M. Qunaibit, and A.-R. Sadeghi, "Losing control: On the effectiveness of control-flow integrity under stack attacks," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, 2015, pp. 952–963.

[20] 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 *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2015.

[21] J. Seibert, H. Okhravi, and E. Söderström, "Information leaks without memory disclosures: Remote side channel attacks on diversified code," in *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, 2014, pp. 54–65.

[22] K. Z. Snow, R. Rogowski, J. Werner, H. Koo, F. Monrose, and M. Polychronakis, "Return to the zombie gadgets: Undermining destructive code reads via code inference attacks," in *Proceedings of the 37th IEEE Symposium on Security & Privacy (S&P)*, May 2016, pp. 954–968.

[23] F. J. Corbató and V. A. Vyssotsky, "Introduction and overview of the Multics system," in *Proceedings of the Fall Joint Computer Conference (AFIPS)*, 1965, pp. 185–196.

[24] D. L. C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural Support for Copy and Tamper Resistant Software," in *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2000, pp. 168–177.

[25] 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 (CCS)*, 2014, pp. 1342–1353.

[26] J. Gionta, W. Enck, and P. Larsen, "Preventing Kernel Code-Reuse Attacks Through Disclosure Resistant Code Diversification," in *Proceedings of the IEEE Conference on Communications and Network Security (CNS)*, 2016, pp. 189–197.

[27] J. Gionta, W. Enck, and P. Ning, "HideM: Protecting the contents of userspace memory in the face of disclosure vulnerabilities," in *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2015, pp. 325–336.

[28] 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 *Proceedings of the 36th IEEE Symposium on Security & Privacy (S&P)*, May 2015, pp. 763–780.

[29] A. Tang, S. Sethumadhavan, and S. Stolfo, "Heisenbyte: Thwarting memory disclosure attacks using destructive code reads," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, 2015, pp. 256–267.

[30] J. Werner, G. Baltas, R. Dallara, N. Otternes, K. Snow, F. Monrose, and M. Polychronakis, "No-execute-after-read: Preventing code disclosure in commodity software," in *Proceedings of the 11th ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2016, pp. 35–46.

[31] K. Braden, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, and A.-R. Sadeghi, "Leakage-resilient layout randomization for mobile devices," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2016.

[32] M. Pomonis, T. Petsios, A. D. Keromytis, M. Polychronakis, and V. P. Kemerlis, "kRˆX: Comprehensive Kernel Protection against Just-In-Time Code Reuse," in *Proceedings of the 12th European conference on Computer Systems (EuroSys)*, 2017, pp. 420–436.

[33] Y. Chen, D. Zhang, R. Wang, R. Qiao, A. M. Azab, L. Lu, H. Vijayakumar, and W. Shen, "NORAX: Enabling execute-only memory for COTS binaries on AArch64," in *Proceedings of the 38th IEEE Symposium on Security & Privacy (S&P)*, 2017, pp. 304–319.

[34] ARM, "Execute-only memory," http://infocenter.arm.com/, 2014.

[35] LWN.net, "Memory protection keys," https://lwn.net/Articles/643797/, 2015.

[36] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "SoK: Automated software diversity," in *Proceedings of the 35th IEEE Symposium on Security & Privacy*, May 2014, pp. 276–291.

[37] F. B. Cohen, "Operating system protection through program evolution," *Computers and Security*, vol. 12, pp. 565–584, Oct. 1993.

[38] S. Forrest, A. Somayaji, and D. Ackley, "Building diverse computer systems," in *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, 1997.

[39] P. Team, "Address space layout randomization," 2003, http://pax.grsecurity.net/docs/aslr.txt.

[40] M. Miller, T. Burrell, and M. Howard, "Mitigating software vulnerabilities," Jul. 2011, http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=26788.

[41] J. Edge, "OpenBSD kernel address randomized link," https://lwn.net/Articles/727697/, 2017.

[42] K. Anand, M. Smithson, K. Elwazeer, A. Kotha, J. Gruen, N. Giles, and R. Barua, "A compiler-level intermediate representation based binary

analysis and rewriting system," in *Proceedings of the 8th European conference on Computer Systems (EuroSys)*, 2013, pp. 295–308.

[43] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz, "Profile-guided automated software diversity," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2013.

[44] S. Crane, A. Homescu, and P. Larsen, "Code randomization: Haven't we solved this problem yet?" in *Proceedings of the IEEE Cybersecurity Development Conference (SecDev)*, 2016, pp. 124–129.

[45] Y. Chen, Z. Wang, D. Whalley, and L. Lu, "Remix: On-demand live randomization," in *Proceedings of the 6th ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2016, pp. 50–61.

[46] D. Williams-King, G. Gobieski, K. Williams-King, J. P. Blake, X. Yuan, P. Colp, M. Zheng, V. P. Kemerlis, J. Yang, and W. Aiello, "Shuffler: Fast and deployable continuous code re-randomization," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 367–382.

[47] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity & randomization for binary executables," in *Proceedings of the 34th IEEE Symposium on Security & Privacy (S&P)*, 2013, pp. 559–573.

[48] E. Shioji, Y. Kawakoya, M. Iwamura, and T. Hariu, "Code shredding: Byte-granular randomization of program layout for detecting code-reuse attacks," in *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC)*, 2012, pp. 309–318.

[49] TechCrunch, "Google says there are now 2 billion active chrome installs," https://techcrunch.com/2016/11/10/google-says-there-are-now-2-billion-active-chrome-installs/, 2016.

[50] M. Conti, S. Crane, T. Frassetto, A. Homescu, G. Koppen, P. Larsen, C. Liebchen, M. Perry, and A.-R. Sadeghi, "Selfrando: Securing the Tor browser against de-anonymization exploits," *PoPETs*, no. 4, pp. 454–469, 2016.

[51] Bulba and Kil3r, "Bypassing StackGuard and StackShield," *Phrack*, vol. 10, no. 56, Jan. 2000.

[52] T. Durden, "Bypassing PaX ASLR protection," *Phrack*, vol. 11, no. 59, Jul. 2002.

[53] 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 *Proceedings of the 36th IEEE Symposium on Security & Privacy (S&P)*, 2015, pp. 745–762.

[54] G. Fresi Roglia, L. Martignoni, R. Paleari, and D. Bruschi, "Surgically returning to randomized lib(c)," in *Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC)*, 2009.

[55] S. J. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. De Sutter, and M. Franz, "It's a TRaP: Table randomization and protection against function-reuse attacks," in *Proceedings of the ACM conference on Computer and Communications Security (CCS)*, 2015, pp. 243–255.

[56] Microsoft, "/ORDER (put functions in order)," 2003, http://msdn.microsoft.com/en-us/library/00kh39zz.aspx.

[57] M. Backes and S. Nürnberger, "Oxymoron: Making fine-grained memory randomization practical by allowing code sharing," in *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[58] "Polyverse," https://polyverse.io/, 2017.

[59] R. N. Horspool and N. Marovac, "An approach to the problem of detranslation of computer programs," *Computer Journal*, vol. 23, no. 3, pp. 223–229, 1980.

[60] R. Wartell, Y. Zhou, K. W. Hamlen, M. Kantarcioglu, and B. Thuraisingham, "Differentiating code from data in x86 binaries," in *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases*, 2011, pp. 522–536.

[61] G. Ramalingam, "The Undecidability of Aliasing," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 5, pp. 1467–1471, September 1994.

[62] M. Ludvig, "CFI support for GNU assembler (GAS)," http://www.logix.cz/michal/devel/gas-cfi/, 2003.

[63] Using the GNU Compiler Collection (GCC), "Common Function Attributes," https://gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html, 2017.

[64] "Profile guided optimization," https://clang.llvm.org/docs/UsersManual.html#profile-guided-optimization.

[65] T. Johnson, "ThinLTO: Scalable and Incremental LTO," http://blog.llvm.org/2016/06/thinlto-scalable-and-incremental-lto.html, 2016.

[66] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "BYTEWEIGHT: Learning to Recognize Functions in Binary Code," in *Proceedings of the 23rd USENIX Security Symposium*, 2014, pp. 845–860.

[67] R. Qiao and R. Sekar, "Function interface analysis: A principled approach for function recognition in COTS binaries," in *Proceedings of the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016.

[68] L. C. Harris and B. P. Miller, "Practical analysis of stripped binary code," *SIGARCH Comput. Archit. News*, vol. 33, no. 5, pp. 63–68, Dec. 2005.

[69] K. ElWazeer, "Deep Analysis of Binary Code to Recover Program Structure," *Dissertation*, 2014.

[70] D. Andriesse, A. Slowinska, and H. Bos, "Compiler-agnostic function detection in binaries," in *Proceedings of the 2nd IEEE European Symposium on Security & Privacy (EuroS&P)*, April 2017, pp. 177–189.

[71] E. Bendersky, "Assembler relaxation," http://eli.thegreenplace.net/2013/01/03/assembler-relaxation, 2013.

[72] Y. Li, "Target independent code generation," http://people.cs.pitt.edu/~yongli/notes/llvm3/LLVM3.html, 2012.

[73] M. Sun, T. Wei, and J. C. Lui, "TaintART: A Practical Multi-level Information-Flow Tracking System for Android RunTime," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, 2016, pp. 331–342.

[74] J. Corbet, "SMP alternatives," https://lwn.net/Articles/164121/, 2005.

[75] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Dynamic reconstruction of relocation information for stripped binaries," in *Proceedings of the 17th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, September 2014, pp. 68–87.

[76] D. Geneiatakis, G. Portokalidis, V. P. Kemerlis, and A. D. Keromytis, "Adaptive Defenses for Commodity Software Through Virtual Application Partitioning," in *Proceedings of the 19th ACM conference on Computer and communications security (CCS)*, 2012, pp. 133–144.

[77] T. Klein, "Relro - a (not so well known) memory corruption mitigation technique," http://tk-blog.blogspot.com/2009/02/relro-not-so-well-known-memory.html, 2009.

[78] "The LLVM Compiler Infrastructure," http://llvm.org.

[79] D. Andriesse, X. Chen, V. van der Veen, A. Slowinska, and H. Bos, "An in-depth analysis of disassembly on full-scale x86/x64 binaries," in *Proceedings of the 25rd USENIX Security Symposium*, 2016, pp. 583–600.

[80] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel, and G. Vigna, "Ramblr: Making Reassembly Great Again," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[81] I. L. Taylor, "Introduction to gold," http://www.airs.com/blog/archives/38, 2007.

[82] S. Kell, D. P. Mulligan, and P. Sewell, "The missing link: Explaining ELF static linking, semantically," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2016, pp. 607–623.

[83] "GNU Binutils," https://www.gnu.org/software/binutils/.

[84] E. Bendersky, "Pure-python library for parsing ELF and DWARF," https://github.com/eliben/pyelftools.

[85] "Protocol Buffers," https://developers.google.com/protocol-buffers/.

[86] S. McCamant, "Large single compilation-unit C programs," http://people.csail.mit.edu/smcc/projects/single-file-programs/, 2006.

[87] "Control flow integrity," https://clang.llvm.org/docs/ControlFlowIntegrity.html.

[88] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, "Control jujutsu: On the weaknesses of fine-grained control flow integrity," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, 2015, pp. 901–913.

[89] R. Rudd, R. Skowyra, D. Bigelow, V. Dedhia, T. Hobson, C. L. Stephen Crane, P. Larsen, L. Davi, M. Franz, A.-R. Sadeghi, and H. Okhravi, "Address-Oblivious Code Reuse: On the Effectiveness of Leakage Resilient Diversity," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[90] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *Proceedings of the 24th USENIX Security Symposium*, 2015, pp. 161–176.

[91] E. Bosman and H. Bos, “Framing signals—a return to portable shellcode,” in *Proceedings of the 35th IEEE Symposium on Security & Privacy (S&P)*, 2014, pp. 243–258.

[92] V. van der Veen, D. Andriesse, M. Stamatogiannakis, X. Chen, H. Bos, and C. Giuffrdia, “The dynamics of innocent flesh on the bone: Code reuse ten years later,” in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, 2017, pp. 1675–1689.

[93] X. Chen, H. Bos, and C. Giuffrida, “CodeArmor: Virtualizing the code space to counter disclosure attacks,” in *Proceedings of the 2nd IEEE European Symposium on Security & Privacy (EuroS&P)*, 2017, pp. 514–529.

[94] M. Zhang, M. Polychronakis, and R. Sekar, “Protecting COTS binaries from disclosure-guided code reuse attacks,” in *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC)*, 2017, pp. 128–140.

[95] B. Randell, “System Structure for Software Fault Tolerance,” in *Proceedings of the International Conference on Reliable Software*, 1975, pp. 220–232.

[96] A. Avizienis, “The n-version approach to fault-tolerant software,” *IEEE Trans. Softw. Eng.*, vol. 11, no. 12, pp. 1491–1501, Dec. 1985.

[97] K. Pettis, R. C. Hansen, and J. W. Davidson, “Profile guided code positioning,” in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 1990, pp. 16–27.

[98] Google, “Syzygy - profile guided, post-link executable reordering,” 2009, http://code.google.com/p/syzygy/wiki/SyzygyDesign.

[99] R. Lavaee and D. Chen, “ABC Optimizer: Affinity Based Code Layout Optimization,” *Technical Report*, 2014.

[100] G. Balakrishnan and T. Reps, “WYSINWYX: What you see is not what you execute,” *ACM Trans. Program. Lang. Syst.*, vol. 32, no. 6, pp. 23:1–23:84, Aug. 2010.

[101] C. Cifuentes and M. V. Emmerik, “Recovery of Jump Table Case Statements from Binary Code,” in *Proceedings of the 7th International Workshop on Program Comprehension (IWPC)*, 1999, pp. 192–192.

[102] X. Meng and B. P. Miller, “Binary code is not easy,” in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 24–35.

[103] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi, “Timely rerandomization for mitigating memory disclosures,” in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, 2015, pp. 268–279.

[104] M. Morton, H. Koo, F. Li, K. Z. Snow, M. Polychronakis, and F. Monrose, “Defeating zombie gadgets by re-randomizing code upon disclosure,” in *Proceedings of the 9th International Symposium on Engineering Secure Software and Systems (ESSoS)*, 2017, pp. 143–160.

[105] Z. Wang, C. Wu, J. Li, Y. Lai, X. Zhang, W.-C. Hsu, and Y. Cheng, “Reranz: A light-weight virtual machine to mitigate memory disclosure attacks,” in *Proceedings of the 13th ACM International Conference on Virtual Execution Environments (VEE)*, 2017, pp. 143–156.

[106] J. Pewny, P. Koppe, L. Davi, and T. Holz, “Breaking and fixing destructive code read defenses,” in *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC)*, 2017, pp. 55–67.

[107] Intel, “System V application binary interface,” https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf, 2013.

[108] “The DWARF debugging standard,” http://dwarfstd.org/.

[109] “LLVM link time optimization design and implementation,” https://llvm.org/docs/LinkTimeOptimization.html.

[110] “The LLVM gold plugin,” http://llvm.org/docs/GoldPlugin.html.

APPENDIX

ADDITIONAL IMPLEMENTATION DETAILS

A. Exception Handling

Our prototype supports the exception handling mechanism that the `x86_64 ABI` [107] has adopted, which includes stack unwinding information contained in the `.eh_frame` section. This section follows the same format as the `.debug_frame` section of DWARF [108], which contains metadata for restoring previous call frames through certain registers. It consists of one or more subsections, with each forming a single CIE (Common Information Entry) followed by multiple FDEs (Frame Descriptor Entry). Every FDE corresponds to a function in a compilation unit. One of the FDE fields describes the `initial_loc` of the function that holds the relative address of the function's entry instruction, which requires to be patched during the rewriting phase.

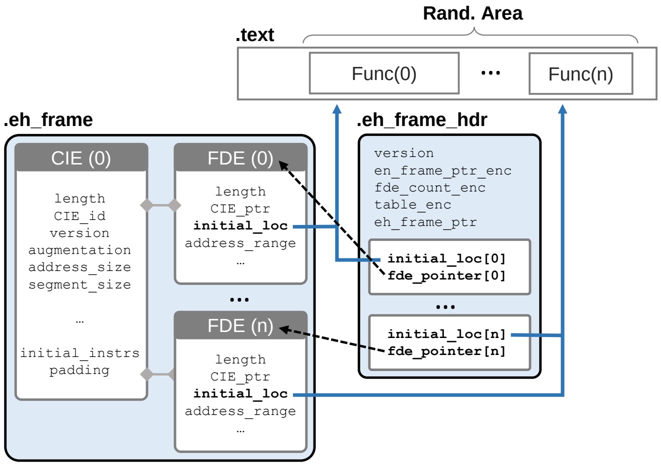


Fig. 8. Structure of an `.eh_frame` section for exception handling. Bold fields must be updated after transformation according to the encoding type specified in the `.eh_frame_hdr` section.

TABLE III
APPLICATIONS USED FOR CORRECTNESS TESTING

Application	Tested Functionality
ctags-5.8	Index a large corpus of source code
gzip-1.8	Compress and decompress a large file
oggenc-1.0.1	Encode a WAV file to OGG format
putty-0.67	Connect to a remote server through the terminal
lighttpd-1.4.45	Start the server and connect to the main page
miniweb	Start the server and connect to the main page
opensshd-7.5	Start an SSH server and accept a connection
vsftpd-3.0.3	Start an FTP server and download a file
libcapstone-3.0.5	Test a disassembly in various platforms
dosbox-0.74	Run an old DOS game within the emulator

As shown in Figure 8, the range an FDE corresponds to is determined by both the `inital_loc` and `address_range` fields. Additionally, `.eh_frame_hdr` contains a table of tuples (`inital_loc, fde_pointer`) for quickly resolving frames. Because these tuples are sorted according to each function's location, the table must be updated to factor in our transformations. Note that our rewriter parses the exception handling sections directly, with no additional information.

Our current CCR prototype does not support randomization with custom exception handling at the basic block level (custom exception handling is fully supported for function-level randomization). As mentioned above, the `.eh_frame` section contains a compact table with entries corresponding to possible instruction addresses in the program. The exception handling mechanism triggers a pre-defined instruction sequence, written in a domain-specific (debugger) language. For example, `DW_CFA_set_loc N` means that the next instructions apply to the first N bytes of the respective function (based on its location). Each FDE may trigger a series of instructions, including the ones in a language-specific data area (LSDA), such as `.gcc_except_table` (if defined), for properly unwinding the stack. To fully support this mechanism, the LSDA instructions should be updated according to the new locations of a functions' basic blocks. We plan to support this feature in future releases of our framework.

B. Link-Time Optimization (LTO)

Starting with v3.9, LLVM supports link-time optimization [109] to allow for inter-module optimizations at link time.[1] Enabling LTO generates a non-native object file (i.e., an LLVM bitcode file), which prompts the linker to perform optimization passes on the merged LLVM IR. Our toolchain interposes at LTO's instruction lowering and linking stage to collect the appropriate metadata of the final optimized code.

C. Control Flow Integrity (CFI)

LLVM's CFI protection [87] offers six different integrity check levels, which are available only when LTO is enabled.[2] The first five levels are implemented by inserting sanitization routines, while the sixth (`cfi-icall`) relies, among other mechanisms, on function trampolines. Our current CCR prototype supports the first five modes, but not the sixth one, because the generated trampolines at call sites are internally created by LLVM using a special intrinsic,[3] rendering their boundaries unknown.

D. Inline assembly

The LLVM backend has an integrated assembler (`MCAssembler`) that emits the final instruction format, which is internally represented by an `MCInst` instance. In general, the instruction lowering process includes the generation of `MCInst` instances. Fortunately, the LLVM assembly parser (`AsmParser`) independently takes care of emitting `MCInst` information also in case of inline assembly, which allows us to tag the parents of all embedded instructions generated from the parser. Moreover, the assembler processes instruction relaxation for inline assembly as needed.

[1] Either `ld.bfd` or `gold` is needed, configured with plugin support [110]. LLVM's LTO library (`libLTO`) implements the plugin interface to interact with the linker, and is invoked by `clang` with the `-flto` option.

[2] Applying CFI requires the `-flto` option at all times. Additionally, both `-fsanitize=cfi-{vcall,nvcall,cast-strict,derived-cast, unrelated-cast, icall}` and `-fvisibility={default,hidden}` flags should be provided to `clang`.

[3] LLVM's `llvm.type.test` intrinsic tests if the given pointer and type identifier are associated.