PointerScope: Understanding Pointer Patching for Code Randomization

Mengfei Xie, Yan Lin, Chenke Luo, Guojun Peng, and Jianming Fu

Abstract—Various fine-grained randomization schemes have been designed to increase the entropy of process space, while none of them can rise from an academic exercise to industrial deployment like Address Space Layout Randomization (ASLR). One of the critical reasons is the incorrectness of randomization caused by the mismatch between their pointer collection capabilities and the high accuracy requirements of the pointer patching task. In this article, we present PointerScope, an accurate compile-time pointer collection scheme deriving from a group of novel observations. The success of PointerScope relies on the complete tracing of the pointer generation process, including the compilation chain from compiler to static linker and the interface specification between them. From this view, PointerScope identifies four types of pointer-related static linker behaviors and clarifies five types of inherent addressing modes in the x86-64 architecture. The vague understanding of them causes the Compiler-assisted Code Randomization (CCR) to incorrectly collect pointers and patch them to the wrong values after randomization. Further, we measure the pointer collection capability of augmented binary analysis, the experimental results show that they can mitigate challenges from the traditional binary analysis by the given premises, but additional heuristics still need to be designed to support the fine-grained randomization.

Index Terms—Code randomization, binary rewriting, pointer patching, addressing mode

1 Introduction

With the widespread deployment of Data Execution Prevention (DEP) in modern operating systems, the exploitation of memory corruption vulnerability has shifted from shellcode injection to code reuse attack, posing a severe threat to vulnerable programs. In response, the automatic software diversity enabled by randomization can break this threat by hiding the real memory code layout and has been researched for over 20 years. ASLR (Address Space Layout Randomization [1]), a coarse-grained randomization scheme, is one of the most representative ones and has been widely deployed in prevalent OSes like Windows, Linux, and iOS, despite facing the information leakage threat and performance overhead of about 5%-10%. However, fine-grained randomization schemes like function-based [2], [3] and basic block-based [4], [5] still stay in the academic research due to runtime crashes triggered by failed pointer patching.

Due to the incompatibility with existing software distribution model and the high cost of producing program variants [6], most existing randomization schemes implement software diversity by static or dynamic binary rewriting at installation, loading, or execution time instead of disrupting the intermediate representation (IR) at compile time. The binary rewriting-based randomization first collects randomization unit boundaries and all cross-references in the binary, then performs the code layout permutation and pointer patching sequentially with the support of these two types of auxiliary information. Moreover, the pointer patching task is particularly sensitive to imprecise information collection, as incorrectly updating even a single pointer can cause the variant to crash at runtime. Unfortunately, it is still a common problem for the static binary analysis to distinguish if a constant in the binary represents a pointer or a data value [5], [7]. To mitigate this mismatch, randomization schemes assisted by the augmented binary analysis require the static linker keeping relocation Sections [3], [8] or the binary being compiled as the position-independent code [5], [9]. Besides, compiler-assisted randomization schemes [1], [4] collect cross-references at compile time and embed metadata in binary to avoid inaccurate binary analysis.

Although previous works have relied on various premises to pursue accurate pointer updating, it is still common for complex programs to crash after the fine-grained randomization. Our experiments show that even allowing the information collection during compilation, CCR [4], a compiler-assisted work, still fails to randomize 17 out of 33 C or C++ programs in SPEC CPU 2017, as well as the complex real-world programs like *GCC* and *FFmpeg*. By manually analyzing these crash cases, we observed that most of them were triggered by a few deviated pointer fixes, which were attributed to two types of pointer mis-collection shown below.

- *Pointer Location Mis-collection*. There can be two types of pointer location mis-collection. The under-collection misses some pointers in binary, causing them not to be updated after the code layout permutation.

- *Mengfei Xie, Chenke Luo, Guojun Peng, and Jianming Fu are with the School of Cyber Science and Engineering, Wuhan University, Wuhan 430072, China. E-mail: {mfxie96, kernelthread, guojpeng, jmfu}@whu.edu.cn.*
- *Yan Lin is with the School of Cyber Security, Jinan University, Guangzhou 510632, China. E-mail: yllinyan@163.com.*

Manuscript received 29 March 2022; revised 24 August 2022; accepted 27 August 2022. Date of publication 31 August 2022; date of current version 11 July 2023.
This work was supported in part by the National Key R&D Program of China (2021YFB3101201), and in part by the National Natural Science Foundation of China (61972297, 62172308, 62172144).
(Corresponding author: Jianming Fu.)
Digital Object Identifier no. 10.1109/TDSC.2022.3203043

In contrast, the over-collection misidentifies constants as pointers, causing them to be rewritten during the pointer patching.

- *Addressing Mode Mis-collection.* Even if pointer locations are correctly collected, misclassifying the addressing modes of a pointer can still make the randomized variant crash. For example, when an absolute pointer is incorrectly treated as a PC-relative pointer, it will be updated from the absolute address to the relative offset between the target address and pointer location.

In this paper, we present a compile-time pointer collection scheme, called PointerScope, which records locations and addressing modes of all pointers in binary by systematically tracing the pointer generation process. Instead of just monitoring pointers emitted at the compiler back-end like other works [4], [7], our study covers the complete compilation chain from compiler to static linker, while also considering the pointer format defined in the interface specification, which allows us to reveal several pointer collection challenges hidden in the compilation process that have been overlooked by previous works [4], [7]. With the accurate pointer information collected by PointerScope, we can implement more reliable compiler-assisted randomization on the one hand, and measure the true pointer collection capability of previous randomization efforts on the other hand.

By coupling the design-level analysis with the manual source-code inspection, PointerScope sheds light on two types of pointer collection challenges hidden in the compilation process, which lead to the mis-collection of pointer locations and addressing modes respectively. First, after recording emitted pointers in the compiler back-end, PointerScope will also revise them during static linking to avoid the pointer location mis-collection, which is because 1) the static linker may update certain compiler-generated pointers or create new ones, and 2) some object files do not have embedded sub-metadata since they are not built by the compiler or pre-compiled by third-party libraries. Second, we challenge the classification of pointer addressing modes adopted by most works [2], [3], [4], [5], which may mismark pointer types and patch the pointers to the wrong value. Through manual code inspection of the compiler back-end and analysis of relocation types defined in the Application Binary Interface (ABI) specification, we fully generalize five inherent addressing modes in the x86-64 architecture, including Absolute addressing, PC-relative addressing, Sym-relative addressing, GOT-relative addressing, and TLS-relative addressing, which support us to patch pointers accurately after locating them.

Benefiting from solving the above two challenges in PointerScope, the improvement and measuring for previous randomization schemes could be performed. We first build up a pointer-diverse dataset, which is compiled separately under different compilation options to cover as many pointer states as possible. Then, we further implement the fine-grained randomization based on the pointer information collected by PointerScope. The experimental results show that it can successfully randomize all programs in SPEC CPU 2017 at function or basic block level. In particular, we also successfully randomized *GCC-7.3.0* and recompiled itself using the randomized compiler. In contrast, CCR [4], the only compiler-assisted fine-grained randomization effort, fails to randomize most complex programs due to the lack of attention to the static linker and the incomplete classification of addressing modes. At last, we evaluate two types of augmented binary analysis. They can partially mitigate the challenges of traditional binary analysis by relying on relocation Sections [3], [8] or position-independent codes [5], [9]. However, additional heuristics are still required to be designed by the corresponding fine-grained randomization work to avoid potential pointer mis-collection.

In a nutshell, we make the following key contributions:

- We systematically dissect the pointer generation process in the compilation chain, with attention to pointers emitted by both the compiler and the static linker, and then generalize their addressing modes. This knowledge can be used to improve the understanding of pointer patching task in existing fine-grained randomization schemes.
- We present a complete compile-time pointer collection scheme, PointerScope. It handles static linker's pointer operations and hybrid compilation during the static linking, and classifies the pointer addressing modes of x86-64 architecture into five categories completely.
- We build pointer-diverse datasets under different compilation options. Evaluation results show that existing fine-grained randomization efforts suffer from the failed pointer patching caused by the mis-collection of pointer locations and addressing modes. PointerScope effectively complements the compiler-assisted randomization scheme and reveals unavoidable information loss in augmented binary analysis.

2 Background and Motivation

This section first briefly describes the pointer patching task of code randomization, whose correctness is guaranteed by the precise collection of pointer locations and addressing modes. Then, we generalize previous randomization schemes and the premises they set up to pursue accurate pointer collection. At last, we discuss the practical deployment barriers faced by the fine-grained randomization, including compatibility, security, performance, and correctness. Most of the barriers have been adequately addressed by previous works while the incorrectness caused by failed pointer patching has remained, which motivates our subsequent study.

2.1 Pointer Patching in Code Randomization

During the compilation process, a pointer can be resolved when its base symbol and target symbol can be determined, which may be performed by the compiler back-end, static linker, or delayed until the program is loading and performed by the dynamic linker. Intuitively, pointer patching is the re-execution of the above process: the binary rewriter resolves all pointers again after the code layout permutation according to the new virtual address of the target and base. As shown in Fig. 1, a complete pointer patching process performs pointer extraction, updating and writing back in sequence with the aid of two types of pointer information: *LOCATION* and *ADDRESSING_MODE*.

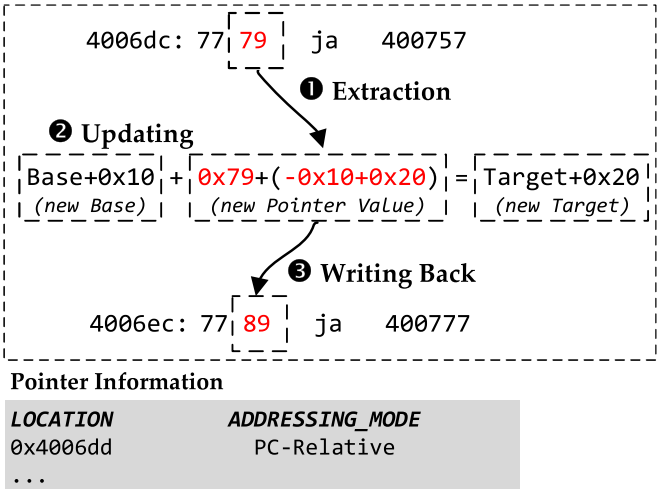


Fig. 1. Patching pointer after the code layout permutation.

Extraction. Pointer extraction step requires the *LOCATION* information previously collected, which is similar to the *r_offset* field in the relocation item. As shown in Fig. 1, the randomization tool first locates the pointer at 0x4006dd, and then it can learn that the initial pointer value is 0x79.

Updating. Pointer updating is performed with the help of *ADDRESSING_MODE* information, similar to the *r_type* field in the relocation item. Intuitively, the offset of pointer value can be interpreted as the change in relative distance between the base address and target address, which can be expressed by Eq. (1): The *Base* and *Target* represent the virtual address of the base and target, respectively, and the Δb and Δt represent their location offsets after code permutation. Further, it is easy to derive that the new pointer value should add $(-\Delta b + \Delta t)$ to satisfy the equation.

$$\{Base + \Delta b\} + \{Pointer + (-\Delta b + \Delta t)\} = \{Target + \Delta t\} \quad (1)$$

Let us review the pointer in Fig. 1, which is a PC-relative pointer with the next instruction at 0x4006de as its base address, pointing to 0x4006de + 0x79 = 0x400757. After the code layout permutation, the base address and target address are increased by 0x10 and 0x20 respectively, thus the new pointer value is 0x79 + (-0x10 + 0x20) = 0x89.

Writing Back. Similar to pointer extraction, the pointer writing back step requires the *LOCATION* information to determine the address of the write operation: 0x4006ed. Then it encodes the new pointer value into the *ja* instruction in little-endian.

2.2 Binary Rewriting-Based Randomization

Early randomization schemes [10], [11], [12] diversify the intermediate representation at compilation, such as by disrupting the order of functions or inserting random garbage instructions. However, compile-time randomization incurs a significant cost because only one variant can be generated per compilation. Besides, software vendors need to prepare many variants for users to download, which does not conform to the existing software distribution model and lack flexibility.

To avoid the drawbacks of compile-time randomization, most existing works shift to post-compilation randomization and implement software diversity by rewriting the static

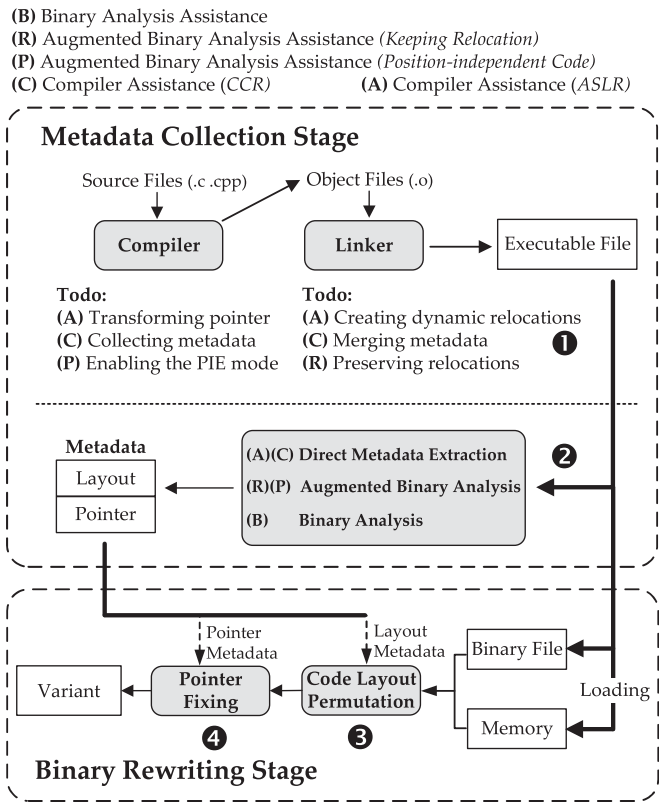


Fig. 2. Overview of binary rewriting-based randomization.

binary file or dynamic process memory space [6]. We refer to such work as binary rewriting-based randomization and split the execution process into two steps: metadata collection and binary rewriting, as shown in Fig. 2. The metadata collection is responsible for collecting auxiliary information such as *LOCATION* and *ADDRESSING_MODE* mentioned in Section 2.1. Then the binary rewriting performs code layout permutation and pointer patching sequentially with the aid of metadata.

Metadata Collection Stage. Table 1 presents three types of metadata collection strategies adopted in existing randomization efforts, which are based on different premises and executed at compile time or post-compilation. Among them, strategies 1 and 2 need to customize the compilation process ❶ to avoid or mitigate the unreliable binary analysis: *compiler-assisted works* collect metadata directly at compile time by extending the compiler and static linker. For example, ASLR [1] transforms pointers to PC-relative addressing at the compiling stage and records the remaining absolute pointers into the dynamic relocation section (*.rela.dyn*) at the static linking stage. CCR [4] collects the metadata from each object file at the compiling stage and merges them into the *.rand* section at the static linking stage. Instead of modifying the compilation toolchain, *augmented binary analysis* only customizes the compilation options, where ASLP [8] and Shuffler [3] ask the static linker to preserve all relocation items after resolving them via the *–emit-relocs* parameter, and other works such as CCFIR [9] and Binigma [5] ask the compiler to output the position-independent code via the *-pie* parameter.

After the compilation, randomization works need to collect layout metadata and pointer metadata from the binary for subsequent binary rewriting stage ❷, where the layout

TABLE 1
Metadata Collection Strategies

	Strategies	Previous Works
1.	Compiler Assistance	*ASLR* [1], *CCR* [4]
2.	Augmented Binary Analysis Assistance	
	• Relocation Assistance	*ASLP* [8], *Shuffler* [3]
	• PIE Assistance	*CCFIR* [9], *Binigma* [5]
3.	Binary Analysis Assistance	*Marlin* [2], *Bin-FR* [13]

metadata records randomization unit boundaries and the pointer metadata records all cross-references. Benefiting from the compile-time collection, *compiler-assisted works* can extract the embedded metadata directly from the binary, such as the *.rela.dyn* section for ASLR [1] and the *.rand* section for CCR [4]. Similarly, *augmented binary analysis-assisted works* [3], [5], [8], [9] run on parameter-specific binaries to achieve more accurate metadata collection. For example, RELOC-assisted works [3], [8] can restore partial pointers from preserved relocation sections, and PIE-assisted works [5], [9] can design customized strategies for position-independent code. In contrast, Marlin [2] and Bin-FR [13] identify code pointers and data pointers by *binary analysis*. This strategy neither involves the compilation process nor relies on any auxiliary information and is therefore suitable for legacy binaries.

Binary Rewriting Stage. This stage is responsible for rewriting the static executable file or dynamic memory space by relying on the two types of metadata collected in the previous stage. Specifically, the binary rewriter first rebuilds the code layout to be randomized with the help of layout metadata, and then permutes them according to specific randomization strategies ❸. Subsequently, the binary rewriter locates pointers to be repaired and then fix up them according to their addressing modes ❹. After the code layout permutation and pointer patching steps, the code layout becomes different while the control-flow and data-flow remain consistent with the original one, thus the variant runs correctly.

2.3 Barriers of Fine-Grained Code Randomization

Although randomization efforts we introduced above span nearly two decades, only the coarse-grained randomization, represented by ASLR [1], has been widely deployed in the real world currently. Our research derives from a reflection on practicality: what are the real barriers that prevent the deployment of fine-grained randomization efforts. In this subsection, we will first discuss four common barriers from compatibility, performance, security, and correctness. Two other trivial but interesting topics, including how developers analyze the remote crash reports sent from variants and how randomization suits the existing software distribution model, will be left for detailed discussion in Section 6.

Incompatibility With the Existing Runtime Mechanisms. Previous works [4], [5] have noted that the fine-grained randomization may break runtime mechanisms such as stack unwinding [14], exception handling [15], and debugging [16]. However, patching their auxiliary information recorded in sections (e.g., *.eh_frame*, *.gcc_except_table*, *.debug*) after randomization can keep these mechanisms working properly in variants.

Conflict With Memory Sharing. Dynamic libraries cannot be shared across processes after the fine-grained randomization. To solve this problem, Oxymoron [17] cuts program code into the smallest sharable piece of memory page size, and then converts all code pointers to indirect references to avoid code page patches after the randomization.

Security. The single-round code randomization was proven to be unsafe as the JIT-ROP attack was proposed [18]. Exploiting the memory disclosure vulnerability, the attacker can dynamically discover and construct gadget chains by performing the recursive code page harvest. In response, several state-of-art defenses can prevent JIT-ROP from special stages: 1) The execute-only memory [19] avoids code leakage by disabling the read permission of code pages. 2) The code pointer hiding [17] prevents recursive code page harvest. 3) The continuous re-randomization [3] limits the building time window of dynamic gadget chains.

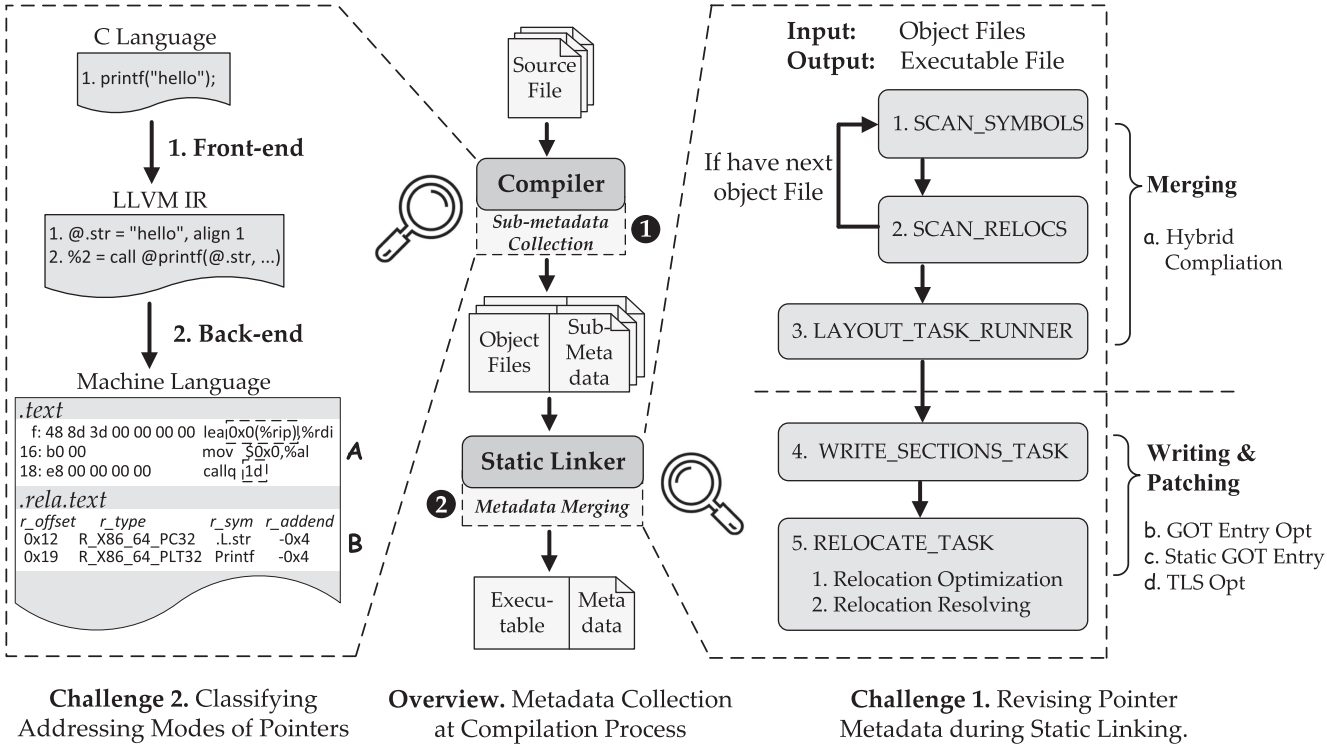
Difficulty in Pointer Patching. Most fine-grained code randomization schemes achieve diversity through binary rewriting, yet neglect the mismatch between the high-precision demands of the pointer patching task and their information collection capabilities. There are nearly a million cross-references in the complex program like *GCC*, failing to update any pointer may cause the variant to crash at runtime.

While previous researches have been proposed to address the compatibility, security, and performance barriers of fine-grained randomization, there is still no practical and complete solution for the accurate pointer patching. The three types of works shown in Table 1 rely on the binary analysis or additional available information to collect pointers and complete fixes, but their design or implementation is still flawed.

For example, randomization works [2], [13] rely on binary analysis to collect pointer information, while many previous researches [7], [20] have confirmed that completely accurate binary analysis is usually impossible. Further, works [3], [5], [8], [9] propose the augmented binary analysis running on parameter-specific binaries. They require keeping relocation sections to provide more available information for randomization, or compiling to position-independent code to simplify analysis. However, how far these preconditions can solve challenges in traditional binary analysis has not been effectively evaluated. Recently, CCR [4] implemented the first fine-grained compiler-assisted code randomization. Even though it is theoretically possible to collect complete pointer information at compile time, we still find that CCR fails to randomize complex real-world programs due to the lack of understanding of the compilation process. Specifically, the lack of monitoring of the static linker causes it to miss some pointers and thus fail to randomize *FFmpeg*, and the incorrect marking of addressing modes causes it to fail to update TLS-relative pointers in *GCC*.

3 Design of PointerScope

In this section, we first briefly describe how PointerScope collects pointers and code boundaries at compile time by extending the compiler and the static linker, which follows the basic



Challenge 2. Classifying Addressing Modes of Pointers

Overview. Metadata Collection at Compilation Process

Challenge 1. Revising Pointer Metadata during Static Linking.

Fig. 3. PointerScope overview and pointer collection challenges in it.

idea of CCR [4]. Importantly, PointerScope identifies two hidden challenges in the pointer collection process, including identifying four types of special cases during static linking and classifying addressing modes of pointers in x86-64 architecture. We will detail these two challenges and corresponding solutions of PointerScope in Sections 3.2 and 3.3, respectively.

3.1 Overview

PointerScope extends the compiler and static linker to collect metadata at compile time for supporting subsequent fine-grained randomization. As shown in Fig. 3 (Overview), PointerScope first monitors each machine code emitted by the compiler back-end and records the function and basic block boundaries as Layout metadata, as well as the cross-references as Pointer metadata ❶. These sub-metadata will be written to the object file waiting for merging as part of the final metadata. Subsequently, PointerScope extracts sub-metadata in object files and merges them according to the splicing order of code fragments, then writes the final metadata to the output binary ❷. Leveraging embedded metadata, a static binary rewriter can randomize the executable at function-level or basic block-level, which does not involve any imprecise binary analysis such as disassembly, function identification, reference analysis, etc.

Going further than CCR, we identified two additional hidden challenges existing in the above metadata collection process, the neglect of which can lead to inconsistency between the collected pointer metadata and the real cross-references, thus confusing the binary rewriter. First, we note that previous works only focus on the compiler-generated pointers, while some special behaviors of the static linker such as updating or creating pointers also need to be concerned. How to discover these special cases hidden in the static linker and revise the pointer metadata in time is the first challenge we face.

Second, we note that most randomization efforts empirically distinguish the addressing modes of x86-64 architecture into three types: Absolute addressing, PC-relative addressing, and jump table. The pointer patching strategies are designed based on them. However, these three types of addressing modes summarized from common cases are incomplete, making them unable to mark pointers emitted by the compiler in some particular modes. How to trace the pointer resolving process across the compilation chain and further comprehensively generalize the addressing modes in x86-64 architecture is the second challenge we face.

We couple design analysis with manual code inspection for the compilation toolchains to address both challenges, allowing PointerScope to avoid failed pointer updating and successfully randomize complex applications in the real world. Design-level analysis can indicate where problems may come from, and subsequent manual code inspections are used to locate specific behaviors and fix them. Both subsequent Sections 3.2 and 3.3 follow this logic.

3.2 *CH1*. Revising Pointer Metadata During Static Linking

Apart from merging sub-metadata in object files, PointerScope also needs to monitor four special behaviors of the static linker and revise pointer metadata in time. As shown in Fig. 3 (Challenge 1), the *gold* static linker performs object file merging and relocation resolving through five *TASK*s. The former is responsible for linking object files into a single binary, and the latter is responsible for establishing the control-flow and data-flow.

However, object files merged by the static linker may be from 1) pre-compiled object files such as crtbegin.o; 2) source

files compiled by LLVM, GCC, etc.; 3) hand-written assembly files. In this case of what we call *Hybrid Compilation (a)*, extending LLVM alone cannot cover all object files involved in the static linking. Besides, relocation items resolved by the static linker may also be various. For example, *gold* tries to optimize relocation items pointing to the *GOT Entry (b)* or *TLS variable (d)* to improve the program performance, which may update partial compiler-generated pointers. Further, we refer to GOT Entries that are resolved at static linking as *Static GOT Entries (c)*, which are created by the static linker but do not have corresponding dynamic relocation items and therefore need to be collected by PointerScope.

3.2.1 Pointer Updating and Increasing in Static Linker

GOT Entry OPT. The GOT mechanism is designed to handle pointers referring to the dynamic link library, whose creation involves the cooperation of the compiler and static linker. As shown in lines#2-#4 of Listing 1, the compiler modifies the pointer with *@GOTPCREL* when it finds that the *dynamic_fun* and *static_fun* symbols are currently undefined. This modifier indicates that the pointer indirectly refers to the target object and requires the static linker to create a GOT Entry for it. Subsequently, when the static linker finds that the *dynamic_fun* symbol is still undefined, it follows the compiler's command to generate the GOT Entry in line#13 as well as the dynamic relocation item in line#17. In contrast, if the target like *static_fun* has been defined in static linking, the static linker will dismiss the compiler's command and optimize the indirect reference back to the direct reference shown in line#9 to improve the runtime performance.

Listing 1. GOT Entry for Undefined Reference

```
 1: # Compiler Output
 2: movq dynamic_fun@GOTPCREL(%rip), %rax
 3: movq static_fun@GOTPCREL(%rip), %rax
 4: push static_fun@GOTPCREL(%rip)
 5:
 6: # Static Linker Output
 7: .text
 8:  795: mov   0x182c(%rip), %rax # 1fc8
 9:  79c: lea   0x15(%rip), %rax   # 7b8
10:  7a3: pushq 0x1827(%rip)       # 1fd0
11:  7b8: ...         # <static_fun>
12: .got
13:  1fc8: 00000000 00000000
14:  1fd0: b8070000 00000000
15: .rela.dyn
16:  r_offset    r_type           r_sym
17:   0x1fc8 R_X86_64_GLOB_DAT dynamic_fun+0
```

By transforming an indirect reference via GOT Entry to a direct reference, the GOT Entry optimization improves runtime performance yet also changes the pointer location. There are two types of GOT Entry optimizations implemented in *gold*'s x86-64.cc file. As shown in Listing 2, the static linker performs GOT Entry optimizations for indirect references under specific opcodes (*mov*, *call*, *jmp*) and specific modifiers (*@GOTPCREL*, *@GOTPCRELX*), where the transformation from line#8 to line#9 will shift the pointer position one byte forward.

Listing 2. GOT Entry Optimization in Gold Static Linker

```
1:  # OPT1. can_convert_mov_to_lea
2:  mov foo GOTPCREL(%rip), %reg
3:    -> lea foo(%rip), %reg.
4:
5:  # OPT2. can_convert_callq_to_direct
6:  callq *foo@GOTPCRELX(%rip)
7:    -> addr32 callq foo
8:  jmpq *foo@GOTPCRELX(%rip)
9:    -> jmpq foo
```

Static GOT Entry. The two types of GOT Entry optimizations implemented in *gold* linker cannot cover all indirect references pointing to defined symbols, which means that there are still a few GOT Entries created by the static linker while being resolved to a specific value on-the-fly. PointerScope needs to collect these Static GOT Entries and revise the final metadata accordingly. There are two types of reasons that can lead to Static GOT Entries. 1) *Special Opcodes.* The *push* instruction escapes the opcode types that GOT Entry optimization handled (*mov*, *call*, *jmp*), thus the static linker still creates the GOT Entry shown in line#14 of Listing 1 and resolves it directly. 2) *Special Modifiers.* In the large code mode, the compiler will replace the *@GOTPCREL* modifier with *@GOT* modifier to fit the larger addressing space, thus the pointer will also not be optimized and is bound to a Static GOT Entry.

TLS OPT. The C language provides a private space for each individual thread through the thread-local storage (TLS), and each reference pointing to the thread-local variable follows one of the below four types of access models: *Local Exec*, *Initial Exec*, *Local Dynamic*, *General Dynamic*, whose performance decrease in order. Similar to the GOT Entry optimization, the static linker checks whether the thread-local storage model chosen by the compiler can be further optimized to the better one. In order to accurately reflect TLS references to thread-local variables in the executable, PointerScope will revise the pointer metadata in time based on the TLS optimization results.

3.2.2 Hybrid Compilation

As shown in Fig. 4, object files involved in static linking may be generated by the compiler (LLVM, GCC, gFortran, etc.), assembler (YASM, etc.), or directly from the pre-compiled library (e.g., CRT), which we refer to as hybrid compilation. In this case, extending only the LLVM compiler cannot cover all sources, leading to the potentially incomplete metadata. In fact, the hybrid compilation is common in ordinary programs, as most of them require the LIBC CRT for initialization and destruction, and hand-written assembly files are necessary for code with architecture-specific or high-performance. Taking *FFmpeg v4.4* as an example, its executable *ffmpeg* consists of 2,002 object files, including 3 object files directly from LIBC and 137 object files from hand-written assembly files processed by YASM.

To adapt to this situation, we could open up new shortcuts by extending every language-translation tool (Compiler

such as LLVM, GCC, gFortran; Assembler such as YASM), but this is an extensive task and cannot handle pre-compiled object files. An alternative idea is filling their missing sub-metadata by disassembly, while this may also introduce imprecision into PointerScope and should be avoided.

In this paper, we propose a hybrid-grained randomization strategy, which ensures correctness while increasing the randomization entropy as much as possible. Specifically, the granularity of the randomization unit is variable: object files containing sub-metadata can participate in code permutation with function-level or basic block-level. For those object files missing sub-metadata, we extract available pointer information from their relocation sections to support the section-level code permutation, which means that the whole *.text* section of these object files will act as an independent randomization cell. It should be noted that although the relocation section records only a subset of pointers in the object file, this is sufficient to support section-level randomization by the following two facts:

- If a pointer can be resolved at the compiler back-end and therefore is not recorded in the relocation section, it must be a relative pointer referring to the inside of its section.
- The internal space of a section does not change after the section-level code permutation, so those missing pointers in relocation sections do not need to be patched.

3.3 *CH2.* Classifying Addressing Modes of Pointers

3.3.1 Addressing Modes in x86-64 Architecture

After locating all pointers emitted by the compiler and static linker, the second challenge we face is to accurately mark the addressing mode for each pointer, which determines whether we can patch them using the correct base address and target address. As Fig. 3 (Challenge 2) shows, high-level language allows developers to refer to memory objects such as functions or data units using symbols directly, while the machine language requires multiple instructions to dereference pointers to the target address. For building this instruction sequence, the compiler back-end first selects proper addressing modes for symbolic references (A), and then tries to resolve pointers to specific values. Pointers that cannot be resolved at compiling time will be recorded in the relocation section (B) to defer resolving at static linking time or dynamic linking time.

During the collaboration of the compilation chain to resolve pointers, the specification of pointer addressing mode is defined by the Application Binary Interface Document (ABI) via Relocation Type, which allows us to recognize at the design level what addressing modes may exist in x86-64 programs. The latest AMD64 ABI 1.0 [21] defines 43 relocation types, which can be classified into Absolute relocation, PC-relative relocation, GOT-relative relocation, and TLS-relative relocation according to the base address they adopt, as shown in Table A.1 (Appendix A, available in the online supplemental material). Among them, the Absolute relocation indicates that the pointer records the target address directly, while the other three kinds of relocation types indicate that pointers take the instruction register *PC*, address of *.got.plt* section, and thread pointer *fs* respectively, to refer to memory objects such as code fragments, data units, GOT Entries, or thread-local variables. Note that we exclude four relocation types in Table A.1 (Appendix A, available in the online supplemental material) since they do not refer to memory objects, including *R_X86_64_NONE*, *R_X86_64_SIZE32*, *R_X86_64_SIZE64*, and *R_X86_64_DTPMOD64*.

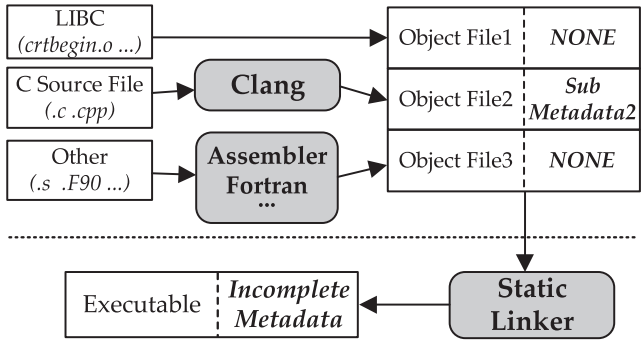


Fig. 4. Hybrid compilation.

To trace forward the generation of relocation type, we further manually inspect the back-end implementation of LLVM, which is responsible for choosing proper addressing modes for pointers. Specifically, the LLVM back-end defines the *MCFixup* object to abstract references in the machine language it will emit, and records the base address through three member variables: *ispcrel*, *modifier*, and *symB*. Where the *ispcrel* variable indicates whether it is a PC-relative pointer or Absolute pointer, and the *modifier* variable indicates whether it is a GOT-relative or TLS-relative pointer. In addition, we observed that LLVM emits a special type of pointer that can take an arbitrary memory object recorded in the *symB* variable as its base address, which we call Sym-relative pointer. In order to keep such a special addressing mode compatible with relocation types currently defined, LLVM will convert the base address from the *symB* object to the PC register when writing Sym-relative pointer to a relocation item, and adjust the addend to balance the equation.

3.3.2 Typical Cases

By coupling the design analysis of relocation type with the manual code inspection of the LLVM back-end, we finally classify the addressing modes inherent in x86-64 architecture into five types, which are selected by the compiler according to multiple factors such as reference distance, reference object, and whether position-independent code. In the following, we will present several typical cases of these five types of addressing modes through Fig. 5, where cases marked by † are wrongly marked as PC-relative pointers by CCR [4], resulting in failed pointer patching.

Absolute Pointer. Absolute pointer often exists in position-dependent executables to point to the data object or the code fragment with an absolute address. As shown in Fig. 5, the machine instruction of line#1 refers to *str1* directly, and line#3 records the absolute address of the basic block *.LBB4_1*.

PC-Relative Pointer. PC-relative pointer takes the instruction register as the base address, and the target object is usually located around itself. A typical case is the control-flow transfer instruction such as *jmp* or *call* as shown in line#4,

Addressing Mode	Assembly Code
Absolute	*// Accesssing data in No-PIE* 1. movabs **$str1**, %rdi *// Jump Table pointer in No-PIE* 2. .LJTI4_0: 3. .quad **.LBB4_1**
PC-relative	*// Transfering control-flow* 4. call **$fun1** *// Accessing data* 5. leaq **$str2(%rip)**, %rax
Sym-relative †	*// Jump Table pointer in PIE* *6. .L8:* *7. .long **.L6 - .L8*** *// Accessing $_GOT_OFFSET_TABLE_ in PIELarge* 8. .L0 9. movq .L0, %rax 10. movq **$_GOT_OFFSET_TABLE_-.L0**, %rcx 11. addq %rcx, %rax
GOT-relative †	12. movq $_GOT_OFFSET_TABLE_, %rax *// Accessing data in PIELarge* 13. movabs **$str3@GOTOFF**, %rbx 14. addq %rax, %rbx *// Accessing GOT Entry in PIELarge* 15. movabs **$fun2@GOT**, %rcx 16. addq %rax, %rcx
TLS-relative †	*// Accessing thread-local variable* 17. movq **$TLSVar@TPOFF**, %rax 18. movl %fs:(%rax), %ecx

Fig. 5. Typical cases of five types of addressing modes. (PIE means the position-independent executable; No-PIE means the position-dependent executable; PIELarge means the position-independent executable with the large code model.)

which calculates the target address by summing PC register and pointer value. In addition, the PC-relative pointer can also be applied in memory access instructions, where the machine instruction in line#5 takes *%rip* as the base address and points to the *str2* object.

Sym-Relative Pointer. Sym-relative pointer can take any memory object in the program as its base and target. A typical case is the pointer in jump table, which takes the head of jump table as its base and points to different code fragments. However, jump tables are not the only place where Sym-relative pointers exist. As the instruction sequence shown in lines#9-#10, which comes from the position-independent binary with the large code model, the pointer takes the basic block *.L0* as the base address and points to the *_GOT_OFFSET_TABLE_*. As far as we know, no randomization work can properly handle such pointers.

GOT-Relative Pointer. GOT-relative pointer takes the *_GOT_OFFSET_TABLE_* (the starting address of the *.got.plt* section) as the base address, and points to the GOT Entry or other memory objects. As the instruction sequence shown in lines#12-#16, the program first stores the *_GOT_OFFSET_TABLE_* (base) to the *%rax* register, then accesses the *fun2*'s GOT Entry and *str3* via the *add* instruction in line#14 and line#16. Note that GOT-relative addressing is essentially a special case of Sym-relative addressing, since it uses the *_GOT_OFFSET_TABLE_* symbol as the base address for pointer dereferencing. Still, we distinguish them in this paper due to their different typical cases.

TLS-Relative Pointer. TLS-relative pointer takes the thread pointer as the base address and points to the thread-local variable. As the instruction sequence shown in lines#17-#18, the program accesses the variable *TLSVar* by summing the *%rax* register and the *%fs* segment register.

4 Implementation of PointerScope

We propose PointerScope, which enables collecting the complete pointer information at compile time by extending the compiler and static linker. Our prototype collects sub-metadata during compiling by modifying *LLVM v10.0.0* [22], then it performs sub-metadata merging and revision during static linking by modifying *gold linker v2.36* of GNU Binutils [23]. Finally, it implements a binary rewriter based on Python 3.6 to randomize programs. Although the design of PointerScope is similar to CCR, we almost completely rewrote the source code of CCR [24] due to different compilation toolchain versions and additional revisions. PointerScope contains about 700 lines of C++ code (within LLVM), 800 lines of C code (within gold linker), and 3500 lines of Python code (binary rewriter) to implement above functions respectively.

4.1 Metadata Collection in LLVM

LLVM takes a single source file as input, it first translates the high-level language into the LLVM intermediate representation on the front-end, then lowers the LLVM IR to the architecture-specific form on the back-end. Finally, it emits the machine code into binary in the final pass - *AsmPrinter*. PointerScope chooses the *AsmPrinter* pass to collect the sub-metadata of each object file and write the results to the *.rand* section due to the following two reasons: 1) All optimizations have been completed before the *AsmPrinter* pass, thus the layout metadata collected at this point can reflect the final code layout. 2) References in the program are converted to hard-coded pointers, thus pointer metadata can be collected.

Listing 3. Pointer Metadata Structure

```
1:  struct PointerTuple {
2:    // Pointer Location
3:    int32_t section;
4:    uint64_t offset;
5:
6:    // Pointer Addressing Mode
7:    int32_t base_section;
8:    uint64_t base;
9:    int32_t target_section;
10:   uint64_t target;
11:  }
```

Specifically, the LLVM back-end abstracts functions and basic blocks into the *MachineFunction* and *MachineBasicBlock* objects, from which PointerScope can gather the boundaries of both. Besides, the LLVM back-end abstracts pointers into

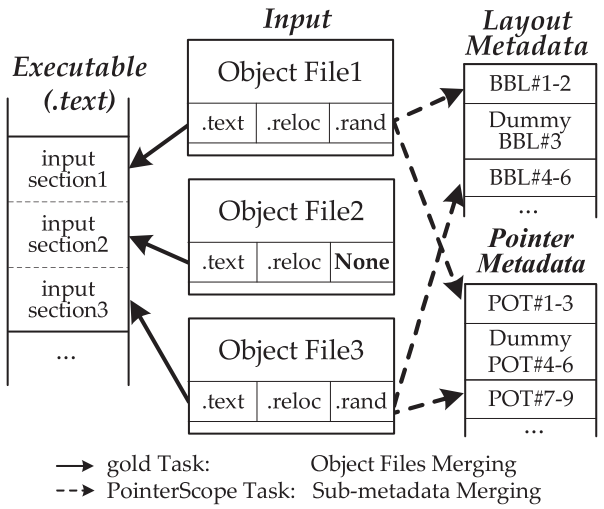


Fig. 6. Metadata merging. (BBL means the basic block information; POT means the pointer information.).

MCFixup objects when it emits machine instructions, from which PointerScope can collect pointer locations and addressing modes and save them in the *PointerTuple* structure shown in Listing 3. Finally, PointerScope will serialize the collected layout metadata and pointer metadata using *protobuf* [25] and write them to the *.rand* section along with the output file.

4.2 Metadata Merging and Revision in Gold

As shown in Fig. 6, the gold static linker is responsible for merging several relocatable object files into the final binary (solid arrows). Meanwhile, PointerScope needs to merge sub-metadata in each object file and revise them in time (dashed arrows). Specifically, PointerScope first merges the layout sub-metadata in terms of the splicing order of *.text* section. Since object file2 misses the *.rand* section, PointerScope will generate DummyBBL#3 for the whole input section of object file2 so that it can participate in code permutation as a unit.

Further, PointerScope merges the pointer sub-metadata and completes the following three revisions: 1) Appending relocation items of object file2 as DummyPOT#4-6. 2) Collecting static GOT Entries into the pointer metadata. 3) Updating the TLS and GOT Entry optimization results into the pointer metadata.

4.3 Binary Rewriter

The binary rewriter of PointerScope is responsible for generating variant programs through code layout permutation and pointer patching, with the assistance of layout metadata and pointer metadata respectively. Specifically, the code layout permutation task first rebuilds the code layout from layout metadata, then performs basic block merging according to constraints such as short pointers and fall-through basic blocks, and finally randomizes the function list as well as the basic code block list of each function. After permutating the code layout, the pointer patching task is responsible for restoring the original control-flow and data-flow in the program. PointerScope first locates all pointers in binary and then updates them according to the change in relative distance between the base and target, which is $(-\Delta b + \Delta t)$ in Eq. (1). Note that the relative distance of Absolute pointers is its target address.

TABLE 2
Programs for Evaluation

Program	Description
SPEC CPU 2017	Compute intensive benchmark.
Real-world Applications	
GCC (v7.3.0)	GNU Compiler for C, C++ and others.
FFmpeg (v4.4)	Format conversion for audio and video.
OpenSSH (v8.8)	Remote connectivity tool with SSH.
Gzip (v1.10)	Data compression program.
Pure-FTPd (v1.0.49)	FTP server.
ProFTPD (v1.3.7)	FTP server.

5 Evaluation

Benefiting from the capability of PointerScope, we can touch complete pointer information within binaries and conduct evaluation from multiple perspectives. After introducing the experiment setup in Section 5.1, we first quantify the distribution of pointer locations and addressing modes under different compilation options, which could be helpful in explaining the differentiated experimental results in later subsections. Then, we use PointerScope to perform fine-grained randomization and detail performance on SPEC CPU 2017 and six real-world applications. Finally, we compare PointerScope's pointer collection results with CCR [4] and two types of augmented binary analysis, respectively, to point out the potential pointer mis-collection that existed in previous works.

5.1 Evaluation Setup

Experimental Platform. Our main experiments were performed on a system equipped with an Intel i7-10700K 3.7GHz CPU, 48GB RAM, running on the x86-64 architecture of Ubuntu 20.04.

Dataset. We build the diverse dataset by expanding open-source programs and compilation options. Our dataset includes all available C and C++ programs from SPEC CPU 2017 and expands with six real-world applications shown in Table 2, considering that the former is compute-sensitive and lacks memory access. Among them, *GCC v7.3.0* is the GNU Compiler program for multiple languages and the *cc1* executable file of it contains more than 681k basic blocks and 1,580k pointers. In addition, the *ffmpeg* executable file in the *FFmpeg v4.4* program contains complex video and audio codecs and it was informed of randomization failure in CCR's GitHub repository. Note that SPEC CPU 2017 already contains an earlier version of *GCC v4.5.0* and customizes it, which is the reason we additionally added *GCC v7.3.0* downloaded from the official site.

Our preliminary experiment shows that the pointer diversity in binary is related to 1) position-independent code, 2) code model [21], 3) target type, and is independent of the optimization level, thus compiling programs with suitable options can help diversify our dataset. The code model indicates the anticipation of the final binary size, including *–mcmodel=small* (default), *–mcmodel=medium*, and *–mcmodel=large*. The large code model will force the compiler to choose

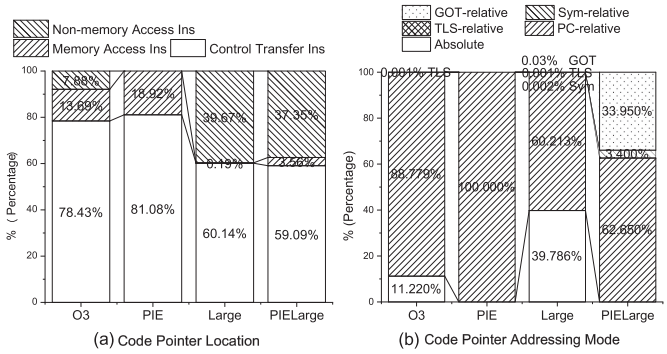


Fig. 7. Location and addressing mode distribution of code pointers.

pointers with a larger addressing space, which is usually used in high-performance computing (HPC). Finally, we combine the following four types of compilation options and use them to compile open-source programs shown in Table 2.

- *O3*. (*-O3 -no-pie –mcmodel=small*)
- *PIE*. (*-O3 -pie –mcmodel=small*)
- *Large*. (*-O3 -no-pie –mcmodel=large*)
- *PIELarge*. (*-O3 -pie –mcmodel=large*)

Completeness Guarantee. Our experiments take the pointer metadata collected by PointerScope as the ground truth, which records each pointer's location and addressing mode in struct *PointerTuple* shown in Listing 3. The completeness guarantee of PointerScope relies on the manual inspecting source code from compiler back-end to static linker, as well as the randomization evaluation with best effort: PointerScope not only successfully randomized all programs in the SPEC CPU 2017 suite, but also randomized six complex real-world applications including *GCC v7.3.0* and *FFmpeg v4.4*. We will detail these randomization results in Section 5.3.

5.2 Pointer Distribution With Different Compilation Options

In this subsection, we respectively present the distribution of code pointers and data pointers in terms of their locations and addressing modes. The results indicate the existence of compiler's pointer selection preferences under different compilation options, which further shape the pointer statistical characteristics in different binary sets and concretize challenges faced by the traditional binary analysis.

5.2.1 Code Pointer Distribution

Fig. 7a shows that the code pointers are distributed in three types of instructions, including control-flow transfer instructions, memory access instructions, and non-memory access instructions. The first two types of code pointers are easy to identify since the immediate number in control flow instructions and memory access instructions must be a pointer. However, the immediate number in non-memory access instructions faces the uncertainty of either pointer or constant. Taking the $mov\ 0x400500,\ \%rax$ instruction as an example, the analyst cannot determine whether 0x400500 represents a virtual address or an integer without considering the context. We further counted the non-memory access instructions containing pointers and found that their opcodes include only the following limited categories requiring additional attention from the reference analysis strategy.

- *MOV*. Transfering the pointer value to the register.
- *ADD*. Calculating the internal element address of a complex memory object.
- *PUSH*. Passing the pointer parameter to the function call.

In particular, no code pointers are located in non-memory access instructions in the PIE binary set, which means that no or very few code pointers are mis-collected in randomization works assisted by PIE. In contrast, there are more code pointers located in non-memory access instructions in the Large binary set and PIELarge binary set, accounting for 39.67% and 37.35%, respectively, suggesting that the reference analysis task in these two types of binary sets needs to pay more attention to distinguishing between pointers and integers. This phenomenon is because the compiler prefers to use less common dereference patterns under Large and PIELarge options, as shown in Listings 4 and 5.

It can be seen that in the large code model, the compiler tends to use absolute pointers of machine word length to refer to function or data objects, thus introducing a lot of non-memory access instructions in line#2 and line#6 of Listing 4. Further, the pointer dereference pattern is more unusual in the position-independent binary with large code model: All memory object references except short jump instructions (line#8 and line#12, Listing 5) take the *_GOT_OFFSET_TABLE_* as their base address, and the virtual address of *_GOT_OFFSET_TABLE_* is calculated by using the basic block where the instruction is located as the base address (line#4, Listing 5), which also introduces a lot of non-memory access *mov* instruction for transferring pointers to registers.

Listing 4. Pointer Dereference Pattern in Large Code Model Binary

```
1:  # data reference
2:  movabsq   $data, %rax
3:  movl      %edx, (%rax)
4:
5:  # function reference
6:  movabsq   $fun, %rax
7:  callq     *%rax
```

Listing 5. Pointer Dereference Pattern in PIE & Large Code Model Binary

```
 1: .L1:
 2:  leaq    .L1(%rip), %rax
 3:  movabsq
 4:      $_GLOBAL_OFFSET_TABLE_-.L1, %rbx
 5:  addq    %rax,  %rbx
 6:
 7:  # data reference
 8:  movabsq    $data@GOTOFF, %rax
 9:  movl       %edx,  (%rbx,%rax)
10:
11:  # function reference
12:  movabsq    $fun@GOTOFF,  %rax
13:  addq       %rbx, %rax
14:  callq      *%rax
```

Fig. 7b shows the distribution of the five types of addressing modes for code pointers. Most code pointers in the O3

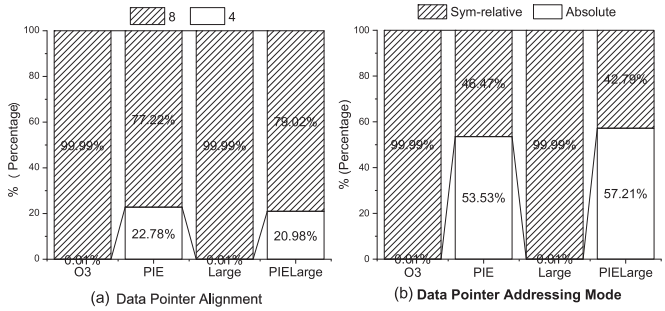


Fig. 8. Location and addressing mode distribution of data pointers.

and Large binary sets choose the Absolute or PC-relative type to address basic blocks, functions, or data objects, and all code pointers are PC-relative addressing in the PIE binary set. Addressing modes in the PIELarge binary set start to be complicated, with 3.4% of Sym-relative pointers and 33.95% of GOT-relative pointers, respectively. As shown in Listing 5 for the pointer dereference pattern under PIELarge option, there is 1 PC-relative pointer (line#2, referring to the basic block), 1 Sym-relative pointer (line#4, referring to *_GOT_OFFSET_TABLE_*), and 2 GOT-relative pointers (line#8 and line#12, referring to the function or data object). Note that there are also 0.03% of GOT-relative pointers and 0.002% of Sym-relative pointers present in the Large binary set, which is due to the fact that a few position-independent object files are involved in the position-dependent binary, giving it both the pointer characteristics present in the PIELarge binary set.

5.2.2 Data Pointer Distribution

Our statistics show that there are only Absolute pointers and jump table pointers (a typical case of Sym-relative pointers) in the data segment, and jump table pointers present two types of formats under different compilation options. In the position-independent binary such as PIE and PIELarge, the jump table consists of Sym-relative pointers of half-machine word length, and these pointers take the jump table header as their base address. Accordingly, it can be observed from the Fig. 8 that PIE and PIELarge binary sets contain 46.47% and 42.79% of 4-bytes length Sym-relative pointers, respectively. In the position-dependent binary such as O3 and Large, the jump table consists of Absolute pointers of machine word length. Accordingly, Fig. 8 shows that 99.99% of data pointers in O3 and Large binaries are 8-byte Absolute pointers. Note that there are still 0.01% of 4-byte length Sym-relative pointers *GCC*, since it forces to introduce some position-independent object files when linking position-dependent *cc1* and *cc1plus*.

5.3 Performance Evaluation

Our dataset for performance evaluation first includes 33 C or C++ programs out of 43 programs provided in SPEC CPU 2017, after removing 8 programs completely written using the Fortran language, 1 program (*627.cam4_s*) that failed to be compiled, and 1 program (*621.wrf_s*) that ran to crash. It should be noted that the exceptions in *627.cam4_s* and *621.wrf_s* are not caused by PointerScope, because we got the same error reporting using the original build toolchain. To further demonstrate the completeness of PointerScope, we also evaluated six real-world applications shown in Table 2, some of which are more complex and thus more challenging than SPEC CPU 2017. Application versions and exact types of activities used for performance evaluation are shown in Table A.2 (Appendix A, available in the online supplemental material). All programs are compiled using the O3 option.

Table 3 shows the performance overhead of PointerScope in terms of compiling, rewriting, and running. Each run was performed 10 times and the average results are reported.

Compiling. The compiling overhead of PointerScope is about 0.89%, which is calculated by the equation $(AllPSTime - AllOrigTime)/AllOrigTime$. This overhead is mainly caused by the metadata collection performed in the compiler and the metadata merging performed in the static linker. In addition, since PointerScope uses *protobuf* [25] to serialize and store metadata, the compiling overhead additionally includes two times metadata compressions (performed in the compiler and static linker) and one time decompression (performed in the static linker).

Rewriting. In the current implementation, the binary rewriting task of PointerScope is performed offline. It can statically rewrite the binary file to generate variants when the system is idle. Therefore, even though Table 3 shows that a complex program like *502.gcc_r* takes about 30s to complete rewriting, this is still acceptable in practice. However, PointerScope can also be extended to a load-time randomization scheme in the future, where the performance of binary rewriting becomes relevant. We believe that the rewriting overhead can be optimized to an acceptable level due to the following two facts: 1) The rewriting time remains approximately linear with the metadata size, which means that the overhead does not increase exponentially with the program complexity. 2) The rewriter is currently a single-thread Python program, and refactoring it using the multi-thread technique and C language can significantly reduce rewriting time.

Running. The average runtime overhead of PointerScope is negligible, only around 0.45% and 0.81% at function reordering and basic block reordering, respectively, due to maintaining a consistent instruction sequence with the original program. Note that programs such as *605.mcf_s* and *657.xz_s* exhibit a slight performance improvement after randomization, while *523.xalancbmk_r* and *623.xalancbmk_s* exhibits an above-average runtime overhead, which is presumably due to the interference of cache locality by code layout permutation [4].

Metadata Size. The space overhead of PointerScope comes entirely from the layout metadata and pointer metadata, as it does not additionally add or remove instructions at compile time. For a program like *502.gcc_r*, which contains 385k code blocks and 749k pointers, the size of collected metadata is 25MB, about half of the binary volume. Note that PointerScope will remove the metadata from variant after the randomization to avoid additional memory overhead and potential information leakage.

Finally, we evaluated the randomization correctness of CCR. The experimental results show that CCR failed to randomize 17 of the 33 programs from the SPEC CPU 2017 suite, with 15 items failing on binary rewriting and 2 items crashing at runtime. Most of the failed programs have more complex code layouts and pointers. Besides, CCR failed to

TABLE 3
PointerScope Performance

Program	Compiling		Rewriting		Running			Metadata Size			CCR Results
	Orig(s)	PS	PS_{FUN}(s)	PS_{BBL}(s)	Orig(s)	PS_{Fun}	PS_{BBL}	BasicBlock	Pointer	MB	
502.gcc_r	44.64	2.44%	25.61	37.79	165.67	1.08%	1.74%	385,877	749,678	25.31	F_{run}
602.gcc_s	44.73	2.44%	25.57	37.58	340.4	1.44%	1.52%	385,855	749,654	25.31	F_{run}
526.blender_r	86.36	1.58%	25.31	32.37	196.23	0.54%	0.77%	353,842	658,396	21.92	F_{re}
510.parest_r	67.09	0.41%	18.68	20.26	330.06	-0.23%	0.00%	373,838	418,869	15.95	F_{re}
521.wrf_r	514.09	0.14%	11.87	12.62	266.38	0.26%	0.41%	11,997	347,819	12.3	F_{re}
607.cactuBSSN_s	21.27	0.85%	9.38	11.87	330.64	0.49%	0.29%	177,926	246,607	8.92	F_{re}
507.cactuBSSN_r	21.36	0.43%	9.41	11.83	215.13	-0.01%	0.41%	177,890	246,311	8.91	F_{re}
523.xalancbmk_r	32	0.85%	9.76	10.59	230.42	5.10%	6.36%	160,175	193,930	7.07	F_{re}
623.xalancbmk_s	32	0.85%	9.77	10.59	241.38	1.21%	5.63%	160,175	193,930	7.07	F_{re}
527.cam4_r	55.82	0.33%	5.97	6.46	186.17	0.31%	1.02%	13,136	194,775	7.02	F_{re}
628.pop2_s	98.09	-0.09%	6.24	6.66	417.01	0.43%	0.37%	12,283	175,623	6.29	F_{re}
500.perlbench_r	10.36	0.00%	5.56	6.95	243.11	0.11%	1.31%	90,442	145,160	5.01	Y
600.perlbench_s	10.27	1.77%	5.57	6.97	311.03	1.28%	1.32%	90,442	145,160	5.01	Y
638.imagick_s	13.82	0.66%	4.53	5.83	426.37	0.76%	-0.03%	65,260	118,721	4.02	Y
538.imagick_r	13.82	0.66%	4.26	5.71	274.9	0.08%	0.12%	63247	113,613	3.86	Y
620.omnetpp_r	13.82	0.66%	4.91	5.38	379.37	0.69%	0.77%	64,032	110,567	3.71	F_{re}
520.omnetpp_s	13.73	0.66%	4.93	5.43	327.36	0.20%	1.65%	64,032	110,567	3.71	F_{re}
511.povray_r	10.45	-0.87%	2.73	3.23	283.18	0.46%	0.16%	32,544	67,020	2.21	F_{re}
508.namd_r	7.27	2.50%	1.42	1.79	211.91	-0.28%	0.08%	22,736	24,430	0.94	F_{re}
525.x264_r	9	0.00%	1.31	1.61	183.33	0.11%	0.05%	17,497	22,408	0.8	Y
625.x264_s	9	0.00%	1.31	1.62	240.32	-0.35%	-0.15%	17,497	22,408	0.8	Y
644.nab_s	1.73	15.79%	1.09	1.36	332.5	0.07%	-0.11%	7,887	14,317	0.49	Y
544.nab_r	1.64	16.67%	1.09	1.35	221.57	-0.20%	0.20%	7,696	13,908	0.48	Y
541.leela_r	3	0.00%	0.85	1.08	337.33	0.28%	-0.42%	6,044	7,316	0.27	F_{re}
641.leela_s	3.09	-2.94%	0.86	1.08	337.64	-0.35%	-0.05%	6,044	7,316	0.27	F_{re}
657.xz_s	2	0.00%	0.92	1.15	548.51	-0.29%	-0.21%	5,242	6,836	0.24	Y
557.xz_r	1.91	0.00%	0.92	1.15	338.01	1.43%	1.63%	5,190	6,727	0.24	Y
631.deepsjeng_s	1.82	-5.00%	0.79	1.02	274.49	-0.26%	0.13%	3,111	5,332	0.18	Y
531.deepsjeng_r	1.82	-10.00%	0.79	1.02	218.4	0.27%	-0.28%	3,111	5,332	0.18	Y
505.mcf_r	1.36	6.67%	0.63	0.84	263.77	-0.63%	0.70%	1,129	1,207	0.05	Y
605.mcf_s	1.36	0.00%	0.63	0.84	516.78	-0.77%	-2.52%	1,129	1,207	0.05	Y
619.lbm_s	1.45	6.25%	0.66	0.86	1,087.87	-0.18%	-0.13%	172	402	0.01	Y
519.lbm_r	1.18	15.38%	0.6	0.81	178.33	-0.93%	-0.19%	156	347	0.01	Y
cc1 *(GCC v7.3)*	195.52	2.26%	58.43	67.69	177.39	2.25%	2.32%	681,967	1,580,004	51.17	F_{re}
cc1plus *(GCC v7.3)*			61.91	71.64				746,863	1,678,422	54.67	F_{re}
ffmpeg	73.57	2.16%	26.14	30.33	5.86	0.38%	0.43%	408,872	598,211	21.27	F_{re}
gzip	0.75	2.92%	0.87	1.10	37.54	0.07%	1.45%	2,621	5,735	0.18	Y
scp *(openssh v8.8)*	3.90	1.27%	2.11	2.51	1.33	0.54%	1.27%	22,403	38,246	1.33	F_{re}
sshd *(openssh v8.8)*			2.21	2.63				24,011	41,018	1.42	F_{re}
pure-ftpd	1.33	1.17%	1.03	1.27	0.49	1.67%	2.29%	3,970	7,236	0.24	F_{re}
proftpd	2.39	3.05%	2.45	2.85	0.50	0.38%	1.10%	24,136	49,430	1.63	Y
Average Overhead		0.89%				0.45%	0.81%				

(*Y* means that the program can be randomized and runs correctly; F_{re} means the program fails at the binary rewriting stage; F_{run} means the program can be randomized but the variant runs to crash. *PS* means PointerScope; PS_{FUN} and PS_{BBL} means the fine-grained randomization at function-level and basic block-level respectively.)

TABLE 4
Pointer Mis-Collection in CCR

CCR		O3	Large	PIE	PIE-Large
Binaries in dataset		57	57	55	55
Object files in dataset		41,846	41,846	37,188	37,188
Pointers in dataset		10,530k	9,129k	6,988k	6,264k
Location Miscollection	*Hybrid Com(obj file)*	2,178	2,178	2,172	2,172
	Static GOT Entry	61	4,796	0	0
	TLS Opt / GOT Opt	43/2,902	0/2,940	0/189,078	0/0
Addressing Mode Miscollection	*Sym-relative (no jump table)*	0	192	0	163,536
	GOT-relative	0	1,805	0	1,631,340
	TLS-relative	84	84	0	0

rewriting 4 real-world applications, including *GCC v7.3.0* (cc1, cc1plus), *FFmpeg v4.4* (ffmpeg), *Openssh v8.8* (scp, sshd) and *Pure-FTPd v1.0.49* (pure-ftpd). We further analyzed the reasons for these rewriting failures, where *Openssh* and *Pure-FTPd* were caused by an implementation error in the CCR's binary rewriter, which adds the base address twice to the offset when calculating the main function address of PIE programs. After fixing this implementation error, CCR can randomize *Openssh* and *Pure-FTPd* successfully. However, the rewriting failures of *GCC* and *FFmpeg* are caused by incorrect pointer collection, with the former misidentifying TLS-relative pointers and the latter ignoring object files from hand-written assembly files, which we will present in detail in Section 5.4.

5.4 Comparison With CCR

To evaluate the mis-collection of pointer locations and addressing modes present in CCR [4], we built pointer-diverse datasets based on four types of compilation options, as shown in Table 4. Each dataset contains millions of pointers, where the missing or misuse of any one of them can lead to the failure of fine-grained randomization. Note that the PIE and PIELarge binary sets miss two executable programs, *cc1* and *cc1plus*, since *GCC* does not support the PIE compilation option. We compare the pointer metadata collected by each of PointerScope and CCR in Table 4. The comparison results are varied in datasets with different compilation options and we will discuss them below.

5.4.1 Pointer Location Mis-Collection in CCR

Hybrid Compilation. As discussed in Section 3.2, object files that comprise an executable may have multiple sources, thus giving the metadata collection capability only for the C compiler would miss some pointers. Apart from the C or C ++ source files, we also observed three other types of sources in the 37k-41k object files that make up the dataset, including 274 hand-written assembly files with .s or .asm suffixes, 1,727 Fortran source files with .f90 suffix, and 177 object files from the Glibc. Among them, the object files from Glibc, such as crt1.o and crtend.o, are present in all programs to initialize or destroy the runtime environment. In addition, the numerical analysis functions in 5 of the 33 C or C++ programs provided by SPEC CPU 2017 are implemented using the Fortran language, and the Gfortran compiler is responsible for generating the corresponding object files. We also noticed that the audio and video codec module of *FFmpeg* is implemented using the architecture-specific assembly language, with 137 of the 2,002 object files that make up the *ffmpeg* executable coming from hand-written assembly files and processed by the YASM assembler.

PointerScope designs the hybrid-grained randomization strategy to suit the above scenario, which means that object files missing sub-metadata will be displaced as separate randomization units (we will call them *huge units* later), so the following security evaluation of these coarse-grained areas is necessary. First, all binaries in our dataset contain pre-compiled object files from Glibc, such as crt1.o and crtbegin.o. However, the average length of these huge units is only 118 bytes, which is even less than a function and thus poses a negligible security threat.

In addition, we found 5 programs contain Fortran-generated object files in the SPEC CPU 2017 suite and 1 real-world application (*FFmpeg*) contains YASM-generated object files. The average length of these huge units is 45,659 bytes and 9,345 bytes, containing an average of 732 and 202 gadgets, respectively. We further evaluate whether these gadgets are sufficient to perform an attack by calculating the *gadget convergence* [26], which refers to whether at least one gadget can be found for each type of Turing-complete (TC) operation. The experimental results show that all huge units do not contain the complete Turing-complete gadget set, which means that it is challenging to construct an available gadget chain even if an attacker can leak one pointer referring to a huge unit. As a supplemental experiment, we also used ROPGadget [27] to automatically build gadget chains for each huge unit, but none of them succeeded.

Static GOT Entry & GOT Entry OPT. Recall the GOT mechanism we introduced in Section 3.2, which consists of two steps: the compiler emits an indirect reference via the GOT Entry, then the static linker performs the GOT Entry optimization. This process is different under four types of compilation options, and we will discuss the differential results shown in Table 4 based on this.

With the PIE compilation option, the compiler tends to convert the undefined reference to the indirect form via a GOT Entry, and the original pointer is modified with *@GOTPCRELX*. If the target object is defined in the subsequent static linking stage, the static linker will optimize the indirect reference back to the direct reference again, which results in 189,078 GOT Entry OPT in the PIE binary set. Besides, all Absolute pointers under the PIE compilation option must have corresponding dynamic relocation items due to the undetermined program base address at loading time, which results in the experimental results of 0 for Static GOT Entry.

With the PIELarge compilation option, the compiler also tends to convert the undefined reference to the indirect form via a GOT Entry, and the original pointer is modified with *@GOT*. This special modifier causes it to escape the GOT Entry optimization of the static linker thus the corresponding experimental results are 0 in Table 4. Besides, the Static GOT Entry in the PIELarge binary set is also 0, for reasons similar to the PIE binary set mentioned above.

With O3 and Large compilation options, the compiler tends to make the pointer directly point to the undefined target symbol instead of converting to the indirect form, thus the number of GOT Entry OPT and Static GOT Entry is theoretically zero. However, a few hard-coded compilation commands in the Makefile may force adding several position-independent object files into the position-dependent binary, resulting in a few pointer distribution features from the PIE compilation option present in the O3 and Large binary set. Further, there are 61 and 4,796 Static GOT Entries in the O3 and Large binary sets respectively, since Absolute pointers in GOT Entries do not need corresponding dynamic relocation items under the determined program base address.

5.4.2 Addressing Mode Mis-Collection in CCR

We present the addressing mode mis-collection of CCR in Section 3.3, it incorrectly marks GOT-relative pointers, TLS-relative pointers, and Sym-relative pointers (non-jump table) as PC-relative pointers, resulting in patching them to the wrong values after the code layout permutation. Among them, we observe 84 TLS-relative pointers in the O3 binary set and the Large binary set respectively, which are both from the *cc1* and *cc1plus*. Listing 6 shows a piece of disassembly instructions from *cc1*, where the constant value 0xFFFFFFD8 is a TLS-relative pointer as it is summed with the *fs* segment to access a thread-local variable. However, CCR incorrectly marks the pointer as the PC-relative addressing and updates it to 0x8CB08. Compared with the original program fetching the value 0 from fs:[0xFFFFFFD8] address and jumping to the branch in line#7, the variant will fetch the non-zero value from fs:[0x8CB08h] address and jump to the branch in line#5 incorrectly.

Moreover, there are 163,536/192 Sym-relative pointers (non-jump table) and 1,631,340/1,805 GOT-relative pointers in the code segments of PIELarge binary set and Large binary set, respectively. All of them come from the pointer dereference pattern shown in Listing 5. In the more common binary sets like O3 and PIE, we have not yet observed the existence of GOT-relative pointers and Sym-relative pointers (non-jump table), which are consistent with the distribution of pointer addressing modes shown in Figs. 7b and 8b.

Listing 6. TLS-Relative Pointer in *GCC v7.3.0*

```
1: 126651D:    mov 0xFFFFFFD8h, %r12
2: 1266524:    mov %fs:%(r12), %rax
3: 1266529:    test %rax, %rax
4: 126652C:    jz short loc_1266568
5: loc_126652E:
6:   ...
7: loc_1266568:
8:   ...
```

5.5 Comparison With Augmented Binary Analysis

Randomization aided by the augmented binary analysis presets the way to compile the binary, such as compiling it to the position-independent code or keeping relocation sections during the static linking. These premises can mitigate the difficulty of binary analysis by preserving available pointer information or making pointers emitted by the compiler more homogeneous. However, a thorough assessment for its reference analysis strategies is still necessary due to the inevitable information loss and the corresponding heuristic strategies. In this subsection, we will discuss separately how far the RELOC-assisted and PIE-assisted randomization can alleviate the challenges of referencing analysis, as well as the shortcomings that still exist. The randomization works we compared are Shuffler [3] and Binigma [5], which are published in top-tier conferences and achieve the fine-grained randomization under RELOC and PIE premises, respectively.

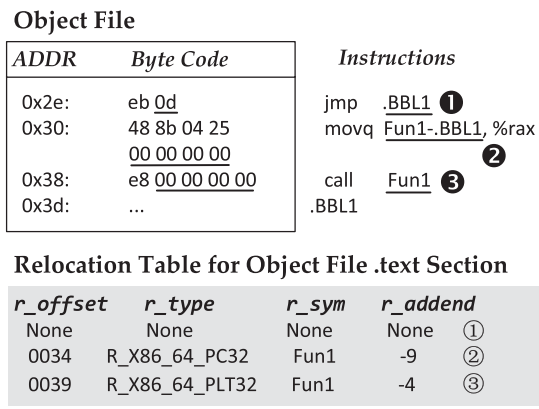


Fig. 9. Three typical types of pointers and their corresponding relocation entries.

5.5.1 RELOC-Assisted Randomization

RELOC-assisted randomization like Shuffler [3] requires the static linker to keep the relocation section after resolving all relocation items, where the *r_offset* field can be used to locate the pointer and the *r_type* field can distinguish its addressing mode. As shown in Fig. 9, the relocation item ③ indicates that the pointer ❸ locates at offset 0x39 of the *.text* section and refers to the *Fun1* symbol via the PC-relative addressing.

However, due to the pointer resolving and base address transformation performed in the compiler back-end, the pointer information recorded in relocation sections is incomplete and ambiguous. Specifically, pointers like $jmp\ .BBL1$ ❶ will not be recorded in the relocation section ① since intra-section references can be resolved directly by the compiler back-end, leading to pointer location mis-collection. Besides, the compiler back-end also tries to equivalently replace the target and base symbol of the pointer in order to minimize the symbol table that needs to be exported. For example, the *r_type* field corresponding to the $Fun1 - .BBL1$ pointer (❷ Sym-relative addressing) is *R_X86_64_PC32* ②. This transformation can lead to ambiguity in relocation types like *R_X86_64_PC32* and lead to the mis-collection of addressing modes, since we cannot distinguish whether it is a PC-relative pointer or a Sym-relative pointer. We will discuss these two issues and the corresponding heuristic strategies below.

Pointer Location Mis-Collection in RELOC. Our experimental results under four types of compilation options show that 1) the relocation section records all data pointers, while 2) misses some code pointers. The first result is easy to understand: Recalling the pointer distribution we described in Section 5.2,

there are two types of data pointers: Absolute pointers and jump table pointers, neither of which can be resolved at the compiler back-end and are therefore recorded in relocation sections to be sent to the static linker. For the second result, we further observe that those missing code pointers due to the intra-section reference only exist in the control-flow transfer instruction and the *lea* instruction, with the percentage distribution shown in Fig. A.1 (Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TDSC.2022.3203043). Guided by this observation, Shuffler [3] only needs to scan for the two specific types of instructions to compensate for code pointers missing in the relocation section.

Addressing Mode Mis-Collection in RELOC. For a Sym-relative pointer, the compiler back-end converts its base address to the pointer location (*PC*) and adjusts the *r_addend* field accordingly to balance the equation. Although this conversion can debloat the exported symbol table, it also leads to PC-relative relocations like *R_X86_64_PC32* being ambiguous: Shuffler needs to distinguish further whether this is a PC-relative pointer or a Sym-relative pointer. According to the pointer distribution described in Section 5.2, the Sym-relative pointers exist only in the position-independent binary's jump table and the PIELarge binary's code pointer. The former has been taken into account by most existing randomization works, including Shuffler [3], and we will discuss it in part B. In particular, we emphasize that jump table pointers are not the only kind of Sym-relative pointer. Others like the code pointers in PIELarge binary should also be given full attention, and heuristic reference analysis strategies should be designed.

5.5.2 PIE-Assisted Randomization

PIE-assisted randomization like Binigma [5] requires the compiler to emit position-independent code, a form in which pointer locations and addressing modes are more homogeneous and easier to analyze. Based on the pointer distribution described in Section 5.2, we will then discuss the extent to which position-independent code can simplify the reference analysis in PIE and PIELarge datasets.

PIE. As shown in Fig. 7, code pointers in the PIE binary set are all located in memory access instructions or control-flow transfer instructions and take *PC* as the base address. Therefore, PIE-assisted randomization works only need to focus on control-flow transfer instructions such as *call* (pointer ❶ in Fig. 10) and *jmp*, and memory access instructions with the $[imm + \%rip]$ pattern (❷), without designing heuristic strategies to distinguish pointers from non-memory access instructions. On the other hand, Fig. 8 shows that there are Absolute pointers and jump table pointers located in the data segment of PIE dataset, where Absolute pointers (❹) are explicitly recorded in the dynamic relocation section (①) to be resolved by the dynamic linker during program loading. In contrast, the jump table pointers (❸) require specific heuristic strategies to distinguish them from data units. A common heuristic designed in Binigma [5] is to match the instruction sequence pattern of accessing the jump table (*INS#0x78b-INS#0x79d*). Our experimental results showed that Binigma could identify all jump tables in the PIE binary set with 100% precision, indicating the reliability of this heuristic strategy.

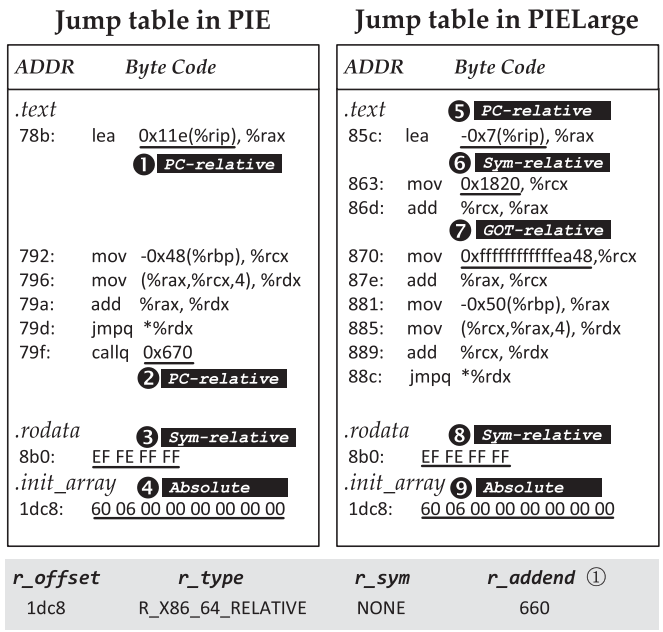


Fig. 10. Typical cases of pointers in PIE and PIELarge programs.

PIELarge. Compared to the PIE binary set, both code pointer and data pointer features under the PIELarge binary set start to become distinct but lack the attention of Binigma [5]. Fig. 7 displays the typical code pointer distribution under the PIELarge binary set, including 62.65% of PC-relative pointers, 33.95% of GOT-relative pointers, and 3.40% of Sym-relative pointers, whose instances correspond to pointers ❺, ❼, ❻, respectively in Fig. 10. The latter two types of pointers are located in non-memory access instructions with *mov* opcode and therefore require the heuristic strategy to identify them. On the other hand, although the data segment of the PIELarge binary set still only includes Absolute pointers (❾) and jump table pointers (❽), the jump table identification algorithm valid in PIE cannot be directly applied to the PIELarge since the instruction sequence pattern (*INS#0x85c-INS#0x88c*) has changed. We tested Binigma on the PIELarge binary set, and our experimental results showed that none of the jump tables were correctly recognized.

6 Discussion

References Analysis for ARM. We have introduced the pointer state displayed in the x86-64 architecture in Section 5.2 from the perspective of pointer position and the five addressing modes. The pointer dereference pattern selected by the compiler in the ARM architecture is similar but more complex for the following two reasons: 1) *Fixed-length instruction.* ARM architecture uses the reduced instruction set with 2 or 4 bytes alignment, resulting in long pointers of 4 or 8 bytes that cannot be stored directly in a single instruction. To resolve the conflict between the large addressing space and fixed-length instructions, the ARM architecture splits and stores long pointers into multiple instructions, called group relocation [28], which results in more diverse pointer formats. An intuitive observation is that there are 124 relocation types in ELF under ARM architecture (AArch64) [29], compared to 45 under x86-64. 2) *High-density instruction encoding.* The ARM architecture uses high-density

instruction coding to fully utilize the limited space in short instructions, which complicates the process of decoding pointers out of instructions and encoding them back. Taking the 4-bytes length *adrp* instruction as an example, it is responsible for storing the high 21 bits of a memory address, where the high 19 bits are stored in the 5th to 24th bit of *adrp*, and the low 2 bits are stored in the 29th and 30th bit of *adrp*. In summary, reference analysis in the ARM architecture will be more challenging than in the x86-64 architecture, which imposes a higher demand on all binary rewriting-based works. We will study this in detail in our future work.

Other Practical Barriers to Randomization. Although randomization enables each program running on the end-user to be unique, it also creates a barrier to analyzing remote crash reports on the developer side. In the current design of PointerScope, the randomized memory layout is entirely reversible with the aid of the seed for mutation. Therefore, dependable local storage and transmission of the mutation seed can be designed with the help of Public Key Infrastructure (PKI): After randomizing a program, the binary rewriter encrypts the mutation seed using the developer's public key and appends it to the variant program. When the variant runs with an exception, end-users report the crash information along with the encrypted mutation seed. Then developers can use their private key to decrypt the mutation seed and further restore the randomized memory layout.

Another practical barrier worth discussing is how to make randomization compatible with the copy-based software distribution model. An intuitive scheme asks users to download programs from the software store and perform randomization by themselves. However, such a randomization scheme is usually only applied to legacy binaries, as inaccurate binary analysis is prone to crashing variants. An alternative is to leave the randomization task to software distributors (e.g., developers or software stores), who can randomly insert NOP instructions or reorder intermediate representations at compile time to achieve accurate randomization. However, this scheme definitely increases the burden on software distributors and breaks the existing software distribution model.

A suitable randomization scheme can be borrowed from the design of ASLR: The randomization task is still assigned to end-users in order to be compatible with the copy-based software distribution model, whereas software distributors need to extract necessary auxiliary information for randomization, including randomization unit boundaries and cross-references, from the compilation process and append them as metadata to the software copy to be distributed. With the aid of metadata, end-users can execute the binary rewriter at any moment to generate variants accurately. For ASLR, this metadata is the dynamic relocation section, which contains absolute pointers that need to be patched after module-grained randomization. For CCR and PointerScope, this metadata is the layout Metadata and pointer Metadata stored in the *.rand* section.

7 Related Work

Randomization. In order to break the series of threats posed by software homogenization, software diversity introduces uncertainty to offer probabilistic protection. Over the past 20 years, researchers have proposed multiple approaches to achieve software diversity from concerns of threat models, security, performance, and practicality [6]. As the first proposed and only widely deployed randomization scheme, ASLR [1] randomizes the process memory layout with module granularity at load time by extending the compiler, kernel, and dynamic linker. However, the entire module can be inferred indirectly from the relative offset when an address leakage exists in the program. Therefore, works [4], [8], [30] are designed with various fine-grained randomization schemes from instruction-level [30], basic block-level [4], and function-level [8].

With the arms race in remote code execution, randomization efforts are also evolving in response to new threat models such as JIT-ROP [18] and Blind ROP [31]. JIT-ROP [18] allows an attacker exploits the memory disclosure vulnerability to dynamically build gadget chains via code page harvest, thus single fine-grained randomization also becomes unsafe. In response, execute-only memory [19] and continuous re-randomization [3] have been proposed to defeat JIT-ROP by limiting code page reading and shortening the attack window, respectively. In addition, Blind-ROP [31] identifies that load-time randomization like ASLR is not compatible with the *Fork* mechanism of the kernel. This means that only the parent process benefits from randomization, while the forked child processes behave with the same memory layout. In response, runtime randomization [32], [33] is proposed to perform secondary randomization when a child process is created. It should be noted that PointerScope does not target any specific randomization scheme, but is based on CCR to study failed pointer patching in randomization.

Disassembly Tools Evaluation. The common challenges in binary analysis are the mixing of code and data and the mixing of pointers and integers. With additional interferences from different architectures, operating systems, and optimization levels, even the commercial disassembly tools like IDA pro are hard to achieve complete correctness [7], and the evaluation of disassembly tools started in that scenario. Works [20], [34] evaluated the capabilities to locate instructions and function boundaries on ARM and x86 architectures, respectively, and the experimental results show that the ability of disassembly tools does not match the assumption of binary-based researches.

Pang et al. [7] evaluated symbolization strategies in nine disassembly tools by collecting pointer information at compile time, which is similar to our study but has the following four differences: (1) *Target.* Unlike disassembly tools that accept a standard binary file as the input, we focus on schemes designed with special premises, such as source code available, relocation prevention, and position-independent binary. (2) *Dimension.* We split the pointer information into two dimensions: location and addressing mode, then evaluate them separately. (3) *Dataset.* Pang's work only measures the impact of different optimization levels, while our dataset additionally considers the impact of the code model and whether the code position is independent, thus can cover more diverse pointer patterns. (4) *Ground Truth.* The ground truth collection tool of Pang's work is implemented based on the open-source code of CCR. Thus same types of pointer mis-collection in CCR also exist in their work, resulting in potential inaccurate experimental results.

Binary Rewriting. Binary rewriting is widely used as a fundamental capability in binary hardening [35], debloating

[36], security analysis [37], and other researches, implemented based on the static reassembly [38], [39] or dynamic binary translation [40]. *Static reassembly* lifts the machine code to the intermediate representation by disassembly and symbolization, then researchers can reorder instructions for randomization [8], [17], [30], [41], insert additional instructions for security policy reinforcement [35], [42] and binary instrumentation [43], [44], or delete existing instructions for debloating [36]. *Dynamic binary translation* works at runtime, which avoids the unreliability of static reassembly but usually comes with a high cost of performance overhead. Note that although our study focuses only on the fine-grained randomization, the results are also applicable to the binary rewriting and all downstream works based on it, since incorrect reference analysis can also cause the variant runs to crash after debloating or instrumenting.

8 Conclusion

Aiming further to facilitate the practical deployment of fine-grained randomization schemes, we scope the failed pointer patching in terms of both pointer location and addressing mode mis-collection. We first present PointerScope, which identifies and addresses two types of potential pointer collection challenges in compile time by dissecting the complete pointer generation process. With the ability to accurately collect pointer information, we used PointerScope to successfully perform fine-grained randomization for SPEC CPU 2017 and real-world applications like *GCC* on the one hand. On the other hand, we used PointerScope to evaluate the pointer collection capability of two types of augmented binary analysis and reveal heuristics that are still necessary. We envision that this knowledge will help researchers to improve the vague understanding of the pointer patching task and provide a foundation for guiding future research.

Availability

The source code of PointerScope is available at https://github.com/whucs303/PointerScope.

Acknowledgments

The authors would greatly appreciate the anonymous reviewers and the associate editor for their valuable feedback.

References

[1] Address space layout randomization, 2003. [Online]. Available: https://pax.grsecurity.net/docs/aslr.txt
[2] A. Gupta, J. Habibi, M. S. Kirkpatrick, and E. Bertino, "Marlin: Mitigating code reuse attacks using code randomization," *IEEE Trans. Dependable Secure Comput.*, vol. 12, no. 3, pp. 326–337, May/Jun. 2015.
[3] D. Williams-King et al., "Shuffler: Fast and deployable continuous code re-randomization," in *Proc. 12th USENIX Symp. Operating Syst. Des. Implementation*, 2016, pp. 367–382.
[4] H. Koo, Y. Chen, L. Lu, V. P. Kemerlis, and M. Polychronakis, "Compiler-assisted code randomization," in *Proc. IEEE Symp. Secur. Privacy*, 2018, pp. 461–477.
[5] S. Priyadarshan, H. Nguyen, and R. Sekar, "Practical fine-grained binary code randomization," in *Proc. Annu. Comput. Secur. Appl. Conf.*, 2020, pp. 401–414.
[6] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "SoK: Automated software diversity," in *Proc. IEEE Symp. Secur. Privacy*, 2014, pp. 276–291.
[7] C. Pang et al., "SoK: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask," in *Proc. IEEE Symp. Secur. Privacy*, 2021, pp. 833–851.
[8] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning, "Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software," in *Proc. 22nd Annu. Comput. Secur. Appl. Conf.*, 2006, pp. 339–348.
[9] C. Zhang et al., "Practical control flow integrity and randomization for binary executables," in *Proc. IEEE Symp. Secur. Privacy*, 2013, pp. 559–573.
[10] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz, "Profile-guided automated software diversity," in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim.*, 2013, pp. 1–11.
[11] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Enhanced operating system security through efficient and fine-grained address space randomization," in *Proc. 21st USENIX Secur. Symp.*, 2012, pp. 475–490.
[12] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis, "kGuard: Lightweight kernel protection against return-to-user attacks," in *Proc. 21st USENIX Secur. Symp.*, 2012, pp. 459–474.
[13] J. Fu, Y. Lin, and X. Zhang, "Code reuse attack mitigation based on function randomization without symbol table," in *Proc. IEEE Trustcom/BigDataSE/ISPA*, 2016, pp. 394–401.
[14] "C++ ABI level I: The unwind process," 2019. [Online]. Available: https://itanium-cxx-abi.github.io/cxx-abi/abi-eh.html#base-abi
[15] C++ ABI level II: C++ ABI, 2019. [Online]. Available: https://itanium-cxx-abi.github.io/cxx-abi/abi-eh.html#cxx-abi
[16] DWARF 4 standard, 2010. [Online]. Available: https://dwarfstd.org/Dwarf4Std.php
[17] M. Backes and S. Nürnberger, "Oxymoron: Making fine-grained memory randomization practical by allowing code sharing," in *Proc. 23rd USENIX Secur. Symp.*, 2014, pp. 433–447.
[18] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *Proc. IEEE Symp. Secur. Privacy*, 2013, pp. 574–588.
[19] S. Crane et al., "Readactor: Practical code randomization resilient to memory disclosure," in *Proc. IEEE Symp. Secur. Privacy*, 2015, pp. 763–780.
[20] 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 *Proc. 25th USENIX Secur. Symp.*, 2016, pp. 583–600.
[21] H. Lu, M. Matz, J. Hubicka, A. Jaeger, and M. Mitchell, "System V application binary interface - AMD64 architecture processor supplement," pp. 588–601, 2022.
[22] LLVM 10.0.0 release notes, 2020. [Online]. Available: https://releases.llvm.org/10.0.0/docs/ReleaseNotes.html
[23] GNU binutils, 2021. [Online]. Available: https://sourceware.org/binutils/
[24] Compiler-assisted code randomization, 2018. [Online]. Available: https://github.com/kevinkoo001/CCR
[25] Protocol buffers - Google's data interchange format, 2008. [Online]. Available: https://github.com/protocolbuffers/protobuf
[26] S. Ahmed, Y. Xiao, K. Z. Snow, G. Tan, F. Monrose, and D. Yao, "Methodologies for quantifying (re-) randomization security and timing under JIT-ROP," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2020, pp. 1803–1820.
[27] ROPgadget - gadgets finder and auto-roper, 2011. [Online]. Available: https://github.com/JonathanSalwan/ROPgadget
[28] ELF for the ARM 64-bit architecture (AArch64), 2022. [Online]. Available: https://github.com/ARM-software/abi-aa/releases/download/2022Q1/aaelf64.pdf
[29] AArch64 ABI release 1.0, 2020. [Online]. Available: https://developer.arm.com/documentation/ddi0487/fc/
[30] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson, "ILR: Where'd my gadgets go?," in *Proc. IEEE Symp. Secur. Privacy*, 2012, pp. 571–585.
[31] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking blind," in *Proc. IEEE Symp. Secur. Privacy*, 2014, pp. 227–242.
[32] K. Lu, W. Lee, S. Nürnberger, and M. Backes, "How to make ASLR win the clone wars: Runtime re-randomization," in *Proc. Annu. Netw. Distrib. Syst. Secur. Symp.*, 2016.
[33] M. Sun, J. Lui, and Y. Zhou, "Blender: Self-randomizing address space layout for android apps," in *Proc. Int. Symp. Res. Attacks Intrusions Defenses*, 2016, pp. 457–480.

[34] M. Jiang, Y. Zhou, X. Luo, R. Wang, Y. Liu, and K. Ren, "An empirical study on ARM disassembly tools," in *Proc. 29th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2020, pp. 401–414.
[35] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Securing untrusted code via compiler-agnostic binary rewriting," in *Proc. 28th Annu. Comput. Secur. Appl. Conf.*, 2012, pp. 299–308.
[36] J. Wu et al., "LIGHTBLUE: Automatic profile-aware debloating of bluetooth stacks," in *Proc. 30th USENIX Secur. Symp.*, 2021, pp. 339–356.
[37] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *Proc. IEEE Symp. Secur. Privacy*, 2015, pp. 709–724.
[38] R. Wang et al., "Ramblr: Making reassembly great again," in *Proc. Annu. Netw. Distrib. Syst. Secur. Symp.*, 2017.
[39] D. Williams-King et al., "Egalito: Layout-agnostic binary recompilation," in *Proc. 25th Int. Conf. Architect. Support Program. Lang. Operating Syst.*, 2020, pp. 133–147.
[40] Pin - A dynamic binary instrumentation tool, 2012. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html
[41] 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 *Proc. 8th ACM SIGSAC Symp. Inf. Comput. Commun. Secur.*, 2013, pp. 299–310.
[42] M. Wang, H. Yin, A. V. Bhaskar, P. Su, and D. Feng, "Binary code continent: Finer-grained control flow integrity for stripped binaries," in *Proc. 31st Annu. Comput. Secur. Appl. Conf.*, 2015, pp. 331–340.
[43] B. Buck and J. K. Hollingsworth, "An API for runtime code patching," *Int. J. High Perform. Comput. Appl.*, vol. 14, no. 4, pp. 317–329, 2000.
[44] O. A. V. Ravnås, "Frida–A world-class dynamic instrumentation framework," 2015. [Online]. Available: https://frida.re/

Mengfei Xie is currently working toward the PhD degree with the School of Cyber Science and Engineering, Wuhan University under the supervision of Dr. Jianming Fu. His current research focuses on system security and software security.

Yan Lin received the PhD degree in computer science from Singapore Management University. She is currently a pre-tenue associate professor with the School of Cyber Security, Jinan University. Her current research focuses on system security, software security, and mobile security.

Chenke Luo is currently working toward the PhD degree with the School of Cyber Science and Engineering, Wuhan University under the supervision of Dr. Jianming Fu. His current research focuses on system security and software security.

Guojun Peng received the PhD degree from Wuhan University, Wuhan, China, in 2008. He is currently a professor with the School of Cyber Science and Engineering, Wuhan University. His research interests include malware analysis and defense, software security, mobile security, and trusted computing.

Jianming Fu received the PhD degree from Wuhan University, Wuhan, China, in 2000. He is currently a professor with the School of Cyber Science and Engineering, Wuhan University. His research interests include system security, software security, AI security, and mobile security.