On the Impact of Exception Handling Compatibility on Binary Instrumentation[†]

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

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

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

Abstract

To support C++ exception handling, compilers generate metadata that is a rich source of information about the code layout. On Linux, this metadata is also used to support stack tracing, thread cleanup and other functions. For this reason, Linux binaries contain code-layout-revealing metadata for C-code as well. Even hand-written assembly in low-level system libraries is covered by such metadata. We investigate the implications of this metadata in this paper, and show that it can be used to (a) improve accuracy of disassembly, (b) achieve significantly better accuracy at function boundary identification as compared to previous research, and (c) as a rich source of information for defeating fine-grained code randomization.

ACM Reference Format:

Soumyakant Priyadarshan, Huan Nguyen, and R. Sekar. 2020. On the Impact of Exception Handling Compatibility on Binary Instrumentation[2]. In *2020 Workshop on Forming an Ecosystem Around Software Transformation (FEAST'20), November 13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3411502.3418428

1 Introduction

Binary instrumentation [6, 11, 21, 34, 39, 41, 45] is a well-established technique for security hardening, application monitoring and debugging, profiling, and so on. Binary instrumentation is more desirable than source-code instrumentation because the vast majority of today's software, including most open-source software, is distributed in binary form. Furthermore, the use of binary-only third-party libraries and hand-written assembly in large software packages make source-based approaches incomplete. Instrumenting all parts of code is critical for many applications, especially those in security, such as CFI [1, 43, 46], SFI [22, 36, 42], program hardening [9, 18, 26, 31, 47], code randomization [5, 14, 25, 29, 38, 40, 44], etc.

Binary instrumentation techniques fall into two broad categories: *dynamic* [6, 21] and *static* [20]. Dynamic instrumentation tools have proved to be robust, but they incur high performance overheads for many applications. Static instrumentation incurs much lower overheads, but has been held back by challenges in accurate disassembly and code pointer identification. With the emergence of position-independent (or relocatable) binaries as the dominant format in recent years, researchers have been able to address these challenges, e.g., in Egalito [41], RetroWrite [11] and SBR [28, 29] systems.

Despite recent advances, deployability of binary instrumentation continues to face significant challenges. One of the major concerns is compatibility. In particular, existing static binary instrumentation tools tend to break stack tracing (for C and C++) as well as C++ exception handling. While compatibility with these features may not be important for proof-of-concept instrumentations, it is hardly a viable option for any software meant for wide deployment.

Although there have been a few research efforts in exception-compatible binary instrumentation (e.g., Zipr++ [15]), most contemporary works [11, 40, 41] tend to dismiss off these compatibility issues as engineering problems. *However, we show in this paper that the impact of stack-tracing and exception compatibility is much more fundamental.* This is because the metadata required to support these features is a rich source of information about the binaries. This information can significantly simplify some aspects of instrumentation (e.g., disassembly and function identification), while introducing new challenges in other aspects (e.g., fine-grained code randomization). We analyze these impacts in this paper, and make the following contributions:

- We show how disassembly can be a challenge for some complex binaries, and discuss how exception handling metadata can help.
- There have been many recent papers on accurate function boundary identification [2, 3, 30, 32]. Many of them rely on complex machine learning or static analysis techniques, yet suffer from significant error rates. In contrast, we show how exception-handling metadata can provide a simple yet far more accurate solution. In particular we can achieve an F1-score of 0.96 by just using the EH metadata, and improve it further to 1.0 with a simple analysis.
- We point out how code randomization techniques can be significantly degraded by exception-handling metadata since it reveals information such as function boundaries, as well as the location of some key instructions that are often targeted in ROP attacks.

Our study is focused on the Linux/x86 platform, with implementation results obtained on 64-bit (x86_64) binaries.

FEAST'20, November 13, 2020, Virtual Event, USA
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8089-8/20/11…$15.00
https://doi.org/10.1145/3411502.3418428

[†]This work was supported by ONR (N00014-17-1-2891) and in part by NSF (CNS-1918667).

2 C++ Exception Handling and Stack Tracing

In C++, exception handling is implemented using try/catch blocks. Catch blocks immediately follow a try block and contain handlers for one or more exceptions that may arise within the try block. If an exception arises outside of a try-block (or if that exception is not handled in the catch blocks associated with the current try-block) then the C++ runtime looks for a try-block in the caller of the current function, or its caller, and so on. Thus, the C++ runtime needs to "walk up the stack" from a callee function to its callers. The C++ compiler generates exception-handling (EH) metadata [24] that is used to perform this *stack unwinding* step.

Stack unwinding may also be needed for generating stack traces when programs crash or experience unrecoverable errors, or when threads exit. On Linux, the same EH-metadata is used to support these features as well. For this reason, EH-metadata is present in Linux for all code[1], not just C++. In fact, most hand-written assembly code in low-level libraries such as `glibc` include stack unwinding information, put in place using the GNU assembler's .cfi directive (which stands for call frame information).

On Linux, EH metadata is stored in the sections *eh_frame*, *eh_frame_hdr* and *gcc_except_table*. Only the first two are needed for stack-unwinding, so the third table is present only for C++ functions. Unlike debugging information that may be present in a binary but not loaded into a process memory, all these sections must be loaded into readable regions of process memory.

The *eh_frame_hdr* section is a binary search table that maps a function to its FDE (Frame description entry), a descriptor for its stack frame. These FDE records are present in *eh_frame* section. FDE records contain special instructions describing how to restore callee saved registers and the stack pointer. These instructions effectively partition the function body into smaller blocks called *unwinding blocks*. Each unwinding block comprises of a set of contiguous instructions that share the same state of the callee-saved registers and the stack pointer.

Consider an instruction that pushes a callee-saved register on the stack. It requires a corresponding restoration operation that will load that register from the stack and then restore the original stack pointer value. Clearly, this is an additional restoration operation that would not have been needed for the preceding instruction. Hence any instruction that changes the stack pointer, including all pushes and pops, results in a new unwinding block. If an instruction saves a callee-saved register to another register or memory, that may also create a new unwinding block. As a result, many unwinding blocks are short, e.g., a push or a pop instruction. This leads to their proliferation, *with a typical function containing about a dozen unwinding blocks* on Linux.

The *gcc_except_table* contains locations of try/catch blocks for every function. Additionally it holds pointers to destructor routines for any object that is created on stack.

In summary, EH metadata contains the following information:

- Function boundary information,
- Pointers to catch routines,
- Pointers to destructor routines of objects created on stack,
- Start address of each *unwinding block*, and
- Arithmetic or load/store operations needed to restore callee saved registers and stack pointers.

[1]The compiler option `-funwind-tables` is enabled by default on Linux/x86. Some projects (e.g., Chromium) override this default option, typically to reduce the space overhead of EH metadata, but naturally, this requires foregoing the use of C++ exceptions.

Application	Library with data within code	Library size
Firefox	libxul.so	126 MB
Chromium, gedit	libffi.so	31 KB
LibreOffice, gedit	libgnutls.so libgcrypt.so	1.4 MB 2.3 MB
gimp, vlc, ssh, evince, apt-get	libgcrypt.so	2.3 MB

Table 1: A few packages with data in the midst of code

3 Disassembly

Accurate disassembly of stripped binaries is a challenging problem for variable-length instruction sets such as those of the Intel x86 architecture. There are two basic techniques for disassembly: linear disassembly and recursive disassembly. Linear disassembly is in general unsound, i.e., it can misclassify data as code. Recursive disassembly can be sound, but is incomplete: it tends to miss code that is only reached via indirect transfers. A number of researchers have proposed heuristics to overcome these drawbacks, but these heuristics are not always successful, and cannot guarantee accurate disassembly in general. Another alternative is exhaustive disassembly that treats every possible offset as an instruction. Multiverse [4] develops such an approach, and has been further improved by Miller et al [23]. However, there is significant overhead in terms of code size as well as runtime overhead, about 60% on SPEC CPU[2].

Unsoundness of linear disassembly stems from the presence of data or padding in the midst of code. Modern compilers such as GCC and Clang have come to avoid inclusion of data in the midst of code, and to use NOPs for padding. This enabled recent binary instrumentation efforts [11, 29, 41] to rely on linear disassembly in their systems. Unfortunately, the assumption about separation of data and code does not always hold, as shown in Table 1. Even without a systematic search, we were able to identify several binaries with embedded data by simply examining a few complex applications such as Firefox and LibreOffice on Linux. Most binary instrumentation tools will fail on these binaries due to incorrect disassembly, or because they transform (and hence corrupt) the data in the midst of code.

In our Secret [44] work, we suggested the use of EH information for recognizing embedded data. But the focus wasn't on disassembly since Secret uses the error-correcting disassembly technique [46] provided by our PSI [45] 32-bit x86 binary instrumentation platform. More importantly, no systematic analysis was undertaken in that work to assess the coverage of EH metadata across a large number of binaries. In this work, we performed such an analysis. We examined each binary in `/bin` and `/lib` on a default 64-bit Ubuntu 18.04 Desktop Linux distribution to determine the

[2]This overhead results from the inability to identify and transform all code pointers at instrumentation time. Instead, code pointers require additional processing at runtime.

percentage of the code that is covered by EH metadata. The average of these percentages across these binaries was 94.7%, showing high coverage. Moreover, as we discuss in the next section, the remaining 5% consists mainly of a few functions that are inserted into every binary by the compiler, which means that almost all of the application-specific code is covered by EH metadata. Thus, EH metadata is a promising source for identifying and marking off data in the midst of code, and designing simple yet robust disassembly techniques that avoid these gaps.

4 Function Identification

Function identification is an essential component of many binary analysis and reverse-engineering tools. It serves as a starting point for recovering other high level program elements such as function parameters and local variables. Many security policy enforcement techniques also operate at function granularity. Other applications of function identification include binary code search [12], binary analysis [33, 37], vulnerability detection [35], and so on.

Recovering functions from stripped COTS binaries is difficult, as much of the symbol and debugging information is lost. In binaries, functions can be defined as a contiguous block of code with one or more entry points and one or more exit points. Function entry points are usually reached by call instructions, except special cases such as a tail call[3]. For direct calls, the target of the call is present in the call instruction itself. The start of functions reached by direct calls can hence be identified by traversing the call graph of a program. However, indirectly reached functions cannot be identified this way.

State-of-the-art approaches for function identification rely on pattern matching, machine learning or static analysis. None of these approaches are 100% accurate and typically result in both misidentifications (false positive) and missed functions (false negatives). Missed functions can affect the coverage of a binary instrumentation tool. Misidentified functions or false positives are often more problematic, and can break the instrumented application, lead to crashes or malfunction. Thus, for robust instrumentation, it is desirable to achieve as close to a zero false positive rate as possible.

4.1 Previous Work on Function Identification

Pattern matching based approaches. Many tools combine call graph traversal with function prologue matching [7, 13, 16, 33] to identify function starts. However, pattern matching in general is not a robust approach and can result in high error rate in terms of both false positives and false negatives. This is because, function prologues/signatures can vary across compilers. Moreover, compiler optimizations may split or reorder the prologue code sequences, thereby degrading the effectiveness of this technique.

Machine learning based approaches. ByteWeight [3] and Shin et al. [32] employ machine learning to identify function starts. By training a model using a large enough set of binaries compiled with multiple compilers, it is possible to improve accuracy across compilers. However, machine learning techniques are never 100% accurate and result in false positives and false negatives. No matter how small the number of misidentifications are, they can affect the usability of any instrumentation tool. Shin et al. achieved a precision of 95%. However, a 5% false positive is too high for practical purposes. Moreover, datasets can be skewed to increase the accuracy rate. Nucleus [2] did an independent validation of datasets used by ByteWeight and found out that many functions were duplicated across training and test datasets, thereby resulting in a better score. When evaluated with a different dataset, ByteWeight's accuracy dropped to 60%.

Static analysis based approaches. *Nucleus* [2] and our previous work FIA [30] rely on static analysis to identify functions. *Nucleus* relies on control-flow analysis to infer indirectly reached functions. It can achieve an accuracy (i.e., F1-score) slightly over 90% across a set of benchmarks. FIA treats any unidentified code region between directly reached functions as a potential function body. It then uses a novel static analysis called function interface analysis to discard most false positives. This improved analysis enables FIA to achieve about 99% accuracy. While this is significantly higher than other approaches mentioned above, a 1% error rate can translate to many false positives and false negatives on large binaries. In combination with the complexity of function interface analysis, this non-negligible error rate prompts researchers to continue to seek techniques that offer a better trade-off between accuracy, complexity and performance.

4.2 Exploiting EH metadata to Identify Functions

Function boundary information is included in EH metadata by default. In particular, function start and size is stored in FDE records in *eh_frame* sections. As pointed out in the previous section, this metadata must cover all functions of a binary and must be present in stripped binaries to support exception handling and stack unwinding at the runtime. Of course, it is possible that real-world binaries may exclude some functions from EH metadata. We have therefore carried out an experimental evaluation that uses EH metadata and compared the results with the previous works described above. In the subsequent sections, we will use the term *EMFI* to refer to function identification using EH metadata.

4.3 Experimental Evaluation

Datasets. We reused the same datasets used in previous works, except that we limited ourselves to 64-bit x86 binaries, as our prototype is currently limited to this architecture. Specifically, ByteWeight and Shin et al. were evaluated on coreutils, binutils and findutils (Dataset 1). FIA uses SPEC CPU 2006 benchmarks (Dataset 2) and GLIBC (Dataset 3) in addition to dataset 1. Note that we did not use the exact same versions of these datasets. Specifically, we used coreutils-8.32, binutils-2.29.1, findutils-4.6.0, SPEC CPU 2017 benchmarks and GLIBC-2.27. We don't expect these version differences to change our results, but they can have a modest impact on some of the previous works, especially those based on machine learning.

Dataset 1 consists of 131 programs written in C and C++. SPEC CPU 2017 consists of 23 programs written C, C++ and Fortran. These programs and glibc were compiled using GCC compiler suite (gcc, g++ and gfortran) on Ubuntu 18.04 (64-bit) operating system,

[3]Tail calls result from a compiler optimization that replaces a call instruction by a jump instruction in the special case where the call instruction just precedes a return.

	Recall		Precision	F1-score	
	EMFI	*EMFI+*	*EMFI & EMFI+*	*EMFI*	*EMFI+*
SPEC	0.9379	1	1	0.9654	1
GLIBC	0.9993	1	1	0.9996	1
Coreutils	0.9371	1	1	0.9669	1
Binutils	0.9897	1	1	0.9941	1
Findutils	0.9407	1	1	0.9692	1

Table 2: Function identification results obtained by exploiting EH metadata

and the resulting binaries used in our analysis. While our experiments were performed on stripped binaries, symbol tables present in unstripped binaries were used for ground truth determination.

Metrics. We use the same *recall*, *precision* and *F1-score* metrics as previous works to measure accuracy. Recall represents the fraction of correctly identified functions and is given by:

$$Recall = \frac{TP}{TP + FN}$$

Here TP stands for true positives and FN stands for false negatives. Precision represents the conditional probability that a function is identified correctly. It is given by:

$$Precision = \frac{TP}{TP + FP}$$

Here FP stands for false positives. F1-score is the harmonic mean of recall and precision and is given by:

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall}$$

Results summary Table 2 shows the accuracy measurement for *EMFI* (i.e., using function boundaries present in EH metadata). By using EH metadata, we are able to achieve zero false positive rates on these datasets. Hence, we achieve a precision of 100%. The recall value shows that on an average we miss 5% of functions.

On closer examination, we realized that the misses were all due to 6 default initialization and clean-up functions inserted into every binary by the compiler. Moreover, pointers to these functions are present in the binary. So, we extended our technique so as to follow function pointers contained in sections of the binaries, specifically, dynamic symbol table and other sections that contain definite code pointers[4]. We follow direct calls recursively from these function bodies. We add any new function found in this manner to our list of identified functions. Note that a new function is added this way only if it is not already covered by EH metadata. We call this enhanced technique as *EMFI+*.

We note that our base *EMFI* technique achieves 100% precision, which is better than all previous approaches. *EMFI*'s F1-score is better than BYTEWEIGHT, Shin et al. and *Nucleus*, but lags FIA on most benchmarks. Our *EMFI+* technique achieves an F1-score of 100% on all datasets, and hence outperforms all previous techniques.

Table 3 compares the function identification scores on Dataset 1 (coreutils, binutils and findutils), which is common to all previous methods. Note that we are simply quoting the numbers from the respective papers [3, 30, 32]. Version differences in the benchmark programs, and especially the compiler, can impact these numbers

[4].init_array and .fini_array contain pointers to initialization and clean up functions of ELF executables

	Recall	Precision	F1-score
BYTEWEIGHT	0.9252	0.9322	0.9287
Shin et al.	0.8991	0.9485	0.9232
FIA	0.9900	0.9912	0.9906
EMFI	0.944	1	0.9706
EMFI+	1	1	1

Table 3: Function identification between different approaches - Dataset 1

	Recall	Precision	F1-score
FIA	0.9861	0.9927	0.9905
Nucleus	0.90	0.97	0.9905
EMFI	0.9379	1	0.9654
EMFI+	1	1	1

Table 4: Function identification comparison with Nucleus and FIA for SPEC

to some extent. To evaluate the effect of compiler versions on our *EMFI* and *EMFI+*, we repeated our experiments with two versions of gcc (gcc-7.5.0 and gcc-4.8.4) as well as llvm-6.0.0. For llvm, we only evaluated coreutils and findutils as binutils is not compatible with this compiler. For *EMFI* we obtained 0 false positives and thereby a precision of 1 across all these compilers. We also observed the same functions (6 in total) missing from the EH metadata across all the compilers. This means that the F1-score of 1.0 achieved by *EMFI+* will be unchanged across these compilers.

5 Code Randomization

Modern Linux distributions apply ASLR to all binaries that are loaded into a process image, including the executable and all libraries. This means that attackers can no longer apriori predict the locations of gadgets they wish to use in a code reuse attack. However, ASLR is considered a relatively weak defense, as a single leaked code pointer can reveal the entire layout of a binary. In response to information leakage, fine-grained code randomization techniques have been proposed [5, 8, 10, 14, 17, 19, 25, 38, 40]. Even if attackers are able to use leaked pointers to determine the base address of a binary, fine-grained code randomization makes it very difficult to predict the locations of the gadgets of interest to them. However, we show that the presence of EH metadata significantly degrades the effectiveness of most of these techniques.

In the rest of this section, we discuss categories of code randomization techniques and how EH metadata degrades their effectiveness. We then suggest some mitigation techniques aimed at restoring their effectiveness. This section assumes that EH metadata is correctly updated after code randomization to ensure stack unwinding compatibility.[5]

Function Reordering. Function reordering is a popular technique used in almost every previous code randomization method. This technique involves permuting the order of functions in a binary. Even with a small binary containing 50 functions, there are 50! possible permutations, yielding a randomization entropy of $\log_2(50!) \approx 214$ bits. Moderate to large binaries yield thousands of bits of entropy, which seems secure against even strong adversaries.

[5]Some code randomization techniques break some of the assumptions underlying stack unwinding, e.g., that function bodies are contiguous. Such techniques need modifications before they can achieve exception compatibility, but a discussion of the specifics is beyond the scope of this paper.

	1 attempt	2 attempts	3 or more attempts
SPEC	47%	19%	34%
Coreutils	58%	31%	11%
Binutils	28%	23%	49%
Findutils	49%	37%	14%

Table 5: Percentage of functions identified by function size in EH metadata

	unwinding blocks with 1 or 2 instructions	unwinding block as `pop; ret;`
SPEC	77%	23%
Coreutils	73%	24%
Binutils	76%	29%
Findutils	72%	22%

Table 6: Percentage of small unwinding blocks and simple gadgets

Some of the earlier techniques [5, 17] rely entirely on function reordering. Many of the more recent techniques [10, 19, 40] combine function reordering with fine-grained code randomization within each function. Even so, many of them derive significant entropy from function reordering.

As we noted before, EH metadata contains (a) the starting location and (b) the size of every function in a Linux/x86 binary. It should also be noted that EH metadata is present in readable memory or else stack unwinding will break. If the attacker targets and leaks this metadata, then they can largely defeat function reordering on its own. This is because function sizes tend to vary considerably, so an attacker can often identify a function just by its size. Table 5 shows the percentage of functions that can be correctly identified within 1, 2 or more attempts using function size information. Identification in one attempt means that the size uniquely identifies a function. Identification in two attempts means that there are just two functions of a given size.

Across these benchmarks, an average of 45% of functions can be uniquely identified from their size. If the attacker focuses his gadget search to this 45% of functions, then, they can carry out their attack as if no function reordering had been done. This is because they can look up the base of address of each of these functions in the EH metadata from their size. Moreover, with just function reordering, the location of every instruction in a function is uniquely determined from its base address. Thus the attacker can determine the location of every gadget of their interest that is located in these 45% of functions.

A simple work-around against this attack is to add random-size gaps within functions so that each function size cannot be related to its original size. This technique introduces a memory overhead in terms of increased code space. This increased code size footprint usually translates into a modest runtime overhead as well, but this increase is unlikely to deter the use of this mitigation. At the same time, it should be noted that this mitigation offers only incomplete protection: for performance reasons, padding size needs to be limited, e.g., it may be limited to less than 100% of code size. While this introduces considerable uncertainty, the size information will still allow an attacker to considerably narrow down the functions that may be located at each of the function base addresses mentioned in EH metadata.

Unfortunately, function sizes, which are present in *eh_frame,* represent the tip of the iceberg in terms of the information an attacker can use to compromise a code randomization technique. Using information in *eh_frame*s, it is possible to defeat more fine-grained randomization techniques, as we discuss below.

Fine-grained Code Randomization. Even if EH metadata is not leaked, function reordering is vulnerable to pointer disclosure attacks. These attacks involve leaking pointer values, e.g., return addresses on the stack. Attackers can then compute gadget locations in the same function as the return address by using the distances between these gadgets and the return address. To thwart such attacks, researchers have proposed many fine-grained code randomizations that involve permuting code blocks (sequences of instructions)[10, 38, 40] or introducing gaps between them [27].

Effectiveness of such fine-grained code randomization is countered by the availability of finer-grained information about code blocks in *eh_frame*s. Specifically, this data identifies the start and size of every unwinding block within a function. Moreover, most of these unwinding blocks are very small. As shown in Table 6, more than 70% of unwinding blocks that have no more than 2 instructions. Moreover, the associated unwinding information specifies key aspects of the semantics of these blocks such as (a) the amount of change to the stack pointer, and (b) the register that is being modified. From this information, the attacker can correctly retrieve location of specific gadgets, e.g., `pop; ret;` or `push; ret;`. *The exact locations of such gadgets can be identified regardless of how much randomization has been done.*

We found that nearly 25% of unwinding blocks are `pop; ret;` gadgets. Thus, an attacker can readily find such gadgets without any search, once she leaks EH metadata.

Another important point to note is that stack pivoting instructions can often be identified from their metadata. Such instructions are often used to corrupt the stack pointer and change it to point to attacker-controlled region of memory. The change happens from loading the stack pointer from memory or another register. Being an instruction that changes the stack pointer, it will be identified as an unwinding block with unwinding information that unmasks its semantics. Although not necessarily mandatory, stack pivoting instructions are very important in practice to launch code reuse attacks, thus often desired by attackers.

In addition to revealing the location of useful instructions and gadgets mentioned above, attackers can also find other gadgets that are very close to these gadgets. Chances are that the randomization technique did not randomize the locations that were just one or two bytes from these instructions. The attacker can hence access such very close-by gadgets to these revealed unwinding blocks.

One way to mitigate such attacks is to introduce gaps in code, together with fake unwinding blocks for these gaps. This technique will confuse the attacker. However, this may not be enough to thwart the attack completely. We recently developed an alternative approach that combines a new exception-compatible high-entropy code randomization technique with optimized EH metadata that exposes much less information [29]. Additional research is needed to systematically explore the full range of options for mitigating the security impact of exception compatibility.

6 Conclusions

To be widely deployable, static binary instrumentation techniques must be compatible with error-handling and reporting mechanisms, such as stack tracing and C++ exceptions. Previous research has tended to dismiss off these compatibility concerns as "engineering." In contrast, we showed that these error-handling features have a much more fundamental impact on binary instrumentation techniques and tools. Specifically, stack-unwinding metadata that underpins these features can provide the basis for improving disassembly accuracy in complex binaries. We also showed that this metadata enables a simple function boundary identification method that is far more accurate than previous methods. On the negative side, we showed that stack unwinding metadata introduces a major point of weakness in fine-grained code randomization. This weakness is most pronounced on the Linux/x86 platform, where stack unwinding information is present for almost every function in every binary. Its impact may be lesser on other platforms, as they may limit unwinding information to just C++ code.

References

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

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

[3] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley. BYTEWEIGHT: Learning to recognize functions in binary code. In *USENIX Security*, 2014.

[4] Erick Bauman, Zhiqiang Lin, and Kevin W Hamlen. Superset disassembly: Statically rewriting x86 binaries without heuristics. In *NDSS*, 2018.

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

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

[7] D. Brumley, I. Jager, T. Avgerinos, and E. Schwartz. Bap: a binary analysis platform. In *Computer Aided Verification*, 2011.

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

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

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

[11] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *IEEE Symposium on Security and Privacy*, 2020.

[12] Halvar Flake. Structural comparison of executable objects. *DIMVA*, 2004.

[13] Laune C Harris and Barton P Miller. Practical analysis of stripped binary code. *ACM SIGARCH*, 2005.

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

[15] Jason Hiser, Anh Nguyen-Tuong, William Hawkins, Matthew McGill, Michele Co, and Jack Davidson. Zipr++ exceptional binary rewriting. In *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation*, 2017.

[16] Hex rays. https://www.hex-rays.com/index.shtml.

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

[18] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure execution via program shepherding. In *USENIX Security Symposium*, 2002.

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

[20] James R. Larus and Eric Schnarr. EEL: machine-independent executable editing. In *Programming language design and implementation*, 1995.

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

[22] Stephen McCamant and Greg Morrisett. Evaluating sfi for a cisc architecture. In *USENIX Security Symposium*, 2006.

[23] Kenneth Miller, Yonghwi Kwon, Yi Sun, Zhuo Zhang, Xiangyu Zhang, and Zhiqiang Lin. Probabilistic disassembly. In *IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019.

[24] James Oakley and Sergey Bratus. Exploiting the hard-working dwarf: Trojan and exploit techniques with no native executable code. In *WOOT*, 2011.

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

[26] Mathias Payer, Tobias Hartmann, and Thomas R. Gross. Safe loading - a foundation for secure execution of untrusted programs. In *S&P*, 2012.

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

[28] Soumyakant Priyadarshan. A study of binary instrumentation techniques. Research Proficiency Report, Secure Systems Lab, Stony Brook University, http://seclab.cs.sunysb.edu/seclab/pubs/soumyakant_rpe.pdf. Accessed: 2020-08-30.

[29] Soumyakant Priyadarshan, Huan Nguyen, and R. Sekar. Practical fine-grained binary code randomization. In *Annual Computer Security Applications Conference (ACSAC)*, 2020.

[30] Rui Qiao and R Sekar. A principled approach for function recognition in COTS binaries. In *Dependable Systems and Networks (DSN)*, 2017.

[31] Rui Qiao, Mingwei Zhang, and R Sekar. A principled approach for rop defense. In *Annual Computer Security Applications Conference*, 2015.

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

[33] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *Security and Privacy (SP)*, 2016.

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

[35] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. Bitblaze: A new approach to computer security via binary analysis. In *International Conference on Information Systems Security*, 2008.

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

[37] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. Ramblr: Making reassembly great again. In *Network and Distributed System Security Symposium*, 2017.

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

[39] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Securing untrusted code via compiler-agnostic binary rewriting. In *ACSAC*, 2012.

[40] David Williams-King, Graham Gobieski, Kent Williams-King, James P Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P Kemerlis, Junfeng Yang, and William Aiello. Shuffler: Fast and deployable continuous code re-randomization. In *OSDI*, 2016.

[41] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P Kemerlis. Egalito: Layout-agnostic binary recompilation. In *ASPLOS*, 2020.

[42] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. In *IEEE Symposium on Security and Privacy*, 2009.

[43] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *IEEE Security and Privacy*, 2013.

[44] Mingwei Zhang, Michalis Polychronakis, and R Sekar. Protecting cots binaries from disclosure-guided code reuse attacks. In *Annual Computer Security Applications Conference*, 2017.

[45] Mingwei Zhang, Rui Qiao, Niranjan Hasabnis, and R Sekar. A platform for secure static binary instrumentation. *ACM VEE*, 2014.

[46] Mingwei Zhang and R Sekar. Control flow integrity for cots binaries. In *USENIX Security*, 2013.

[47] Mingwei Zhang and R Sekar. Control flow and code integrity for cots binaries: An effective defense against real-world rop attacks. In *ACSAC*, 2015.