Zipr++: Exceptional Binary Rewriting

Jason Hiser, Anh Nguyen-Tuong, William Hawkins, Matthew McGill, Michele Co, Jack Davidson
Department of Computer Science, University of Virginia, Charlottesville, VA 22904
{hiser,an7s,whh8b,mm8bx,mc2zk,jwd}@virginia.edu

ABSTRACT

Current software development methodologies and practices, while enabling the production of large complex software systems, can have a serious negative impact on software quality. These negative impacts include excessive and unnecessary software complexity, higher probability of software vulnerabilities, diminished execution performance in both time and space, and the inability to easily and rapidly deploy even minor updates to deployed software, to name a few. Consequently, there has been growing interest in the capability to do late-stage software (i.e., at the binary level) manipulation to address these negative impacts. Unfortunately, highly robust, late-stage manipulation of arbitrary binaries is difficult due to complex implementation techniques and the corresponding software structures. Indeed, many binary rewriters have limitations that constrain their use. For example, to the best of our knowledge, no binary rewriters handle applications that include and use exception handlers—a feature used in programming languages such as C++, Ada, Common Lisp, ML, to name a few.

This paper describes how Zipr, an efficient binary rewriter, manipulates applications with exception handlers and tables which are required for unwinding the stack. While the technique should be applicable to other binary rewriters, it is particularly useful for Zipr because the recovery of the IR exposed in exception handling tables significantly improves the runtime performance of Zipr'ed binaries—average performance overhead on the full SPEC CPU2006 benchmark is reduced from 15% to 3%.

1 INTRODUCTION

Software systems are a vital component of critical infrastructure such as transportation systems, communications systems, financial systems, power generation and distribution systems, and defense systems. Current software development methodologies and practices, while enabling the rapid production of these complex software systems, can have a serious negative impact on software quality. These negative impacts include excessive and unnecessary software complexity, higher probability of software vulnerabilities, diminished execution performance in both time and space, and the inability to easily and rapidly deploy even minor updates to deployed software, to name a few. Furthermore, the move to network and mobile computing and constant pressure for new features and capabilities has made these vulnerable systems easily accessible to malicious adversaries.

Consequently, there has been growing interest in developing capabilities enabling late-stage software manipulation to address these negative impacts. For example, there is high interest is modifying legacy binaries to improve security. Modification techniques suggested include adding security constructs (i.e., control-flow integrity, canaries, diversity, etc.), reducing the attack surface by removing unneeded functionality, and applying patches to fix security issues discovered post-deployment. Unfortunately, highly robust, late-stage manipulation of arbitrary binaries to apply systemic changes is difficult due to complex implementation techniques and the corresponding software structures. Indeed, many binary rewriters have limitations that constrain their use. To the best of our knowledge, no static binary rewriters handle applications that use exception handlers or stack unwinding—a feature integral to programming languages such as C++, Ada, Common Lisp, ML, to name a few.

This paper describes how we have extended Zipr [5], an efficient binary rewriter, to manipulate applications with exception handlers and tables, which are required for unwinding the stack. Unwinding the stack is necessary for handling exceptions, object-oriented programs (invoking necessary destructors), multi-threading (when a thread exits, destructors on the stack must be invoked), and to support debuggers such as gdb, dbx, and lldb. While the approach should be applicable to other binary rewriters, it is particularly useful for Zipr because the recovery of the IR in exception handling tables significantly improves the runtime performance of Zipr'ed binaries—average performance overhead on the full SPEC CPU2006 benchmark is reduced from 15% to 3%.

The major contributions of this paper are:

- We highlight the importance and costs of supporting stack unwinding for exception handling in a static binary rewriter.
- We propose the first technique for constructing and managing an easy-to-use intermediate representation (IR), as well as realizing that IR into a rewritten form in an output binary.
- We thoroughly evaluate the proposed technique in a modern, fully-featured binary rewriter, including evaluation on the full SPEC CPU2006 benchmark suite, a case study highlighting a security transformation, and a case study demonstrating stability across programming languages (C++, Ada) and supporting runtime environments (Ada).

2 EXCEPTIONAL BINARY REWRITING

2.1 Architectural Overview

Zipr++ provides full support for exception handling by extending the Zipr binary rewriting infrastructure. Zipr is a platform for statically rewriting programs/libraries without access to their source code, debugging symbols or other metadata and offers an API and

FEAST'17, November 3, 2017, Dallas, TX, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5395-3/17/11…$15.00
https://doi.org/10.1145/3141235.3141240

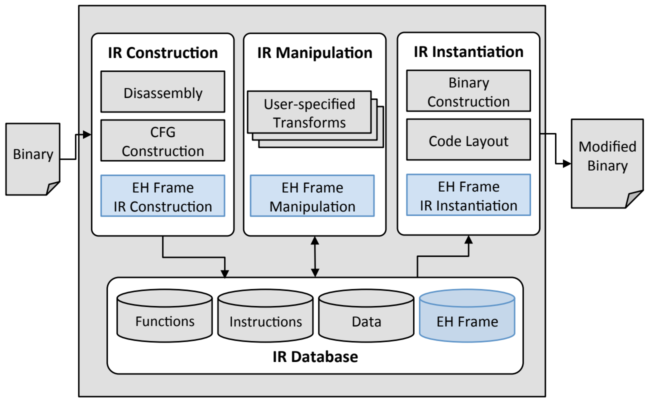

Figure 1: Overview of the Zipr++ rewriting pipeline. New extensions to support exception handling are shown in blue.

SDK for third parties to write extensions and customizations to instantiate arbitrary user-defined transformations [5]. Although it is a tool for general static binary rewriting, the primary use case for Zipr to date has been to retrofit legacy binaries with enhanced security. Zipr was the key defensive technology used in the DARPA Cyber Grand Challenge to diversify and augment binaries with point patches and control-flow integrity protections [5, 14]. Other uses include effecting moving-target defenses against blind ROP attacks [6] and protecting autonomous drones [4].

Figure 1 illustrates the three major phases of the Zipr/Zipr++ pipeline: Intermediate Representation (IR) Construction, IR Manipulation and IR Instantiation, with the required extensions to support exception handling (shown in blue).

In the IR Construction phase, Zipr analyzes a program or library to detect instructions, functions and data objects.

The recovered IR is passed to the IR Manipulation stage where user-defined transformations programmatically modify the input program by altering its IR.

Finally, the transformed IR is passed to the IR Instantiation phase that generates a rewritten version of the program (Section 2.4). The transformed program is executable on the same platform as the original program without any additional runtime support.

To generate an efficient statically rewritten program/library, IR Instantiation relies on the freedom to reassemble most instruction sequences (basic blocks, functions, etc.) in the output at different addresses than their addresses in the input. There are certain sequences that cannot be moved, however. The IR Construction phase detects these immovable sequences and *pins* their addresses. For example, if a return address is used for EH-driven stack unwinding, Zipr must pin that return address, and a call instruction that writes the address to the stack must be updated to write the pinned address. We extended this functionality (in Zipr++) by supporting EH-table rewriting, allowing the tool more flexibility in placement of functions and the opportunity to generate more optimized code sequences (Section 2.2).

2.2 EH Frame IR Construction

In Linux ELF executable files, the exception handling and stack unwinding information is stored in several sections, namely `.eh_frame_hdr`, `.eh_frame`, and `.gcc_except_table`, and some is stored as read-only data and code in the `.text` section.

The main entry point for stack unwinding and exception handling is the `.eh_frame`. This section contains a sequence of variable length table entries. A table entry is either a *common information entry* (CIE) or a *frame descriptor entry* (FDE). An FDE describes how to unwind and cleanup the stack for a range of instructions. Each FDE points at a CIE, which contains information common across many FDEs. CIEs are a way to avoid duplication in the FDEs and save space in the tables. It is common to have one FDE for each function in the program, but a function may have multiple FDEs for different portions of the program. Figure 2 shows an example, with `FDE 3` expanded.

FDEs record a variety of information necessary for stack unwinding, cleanup, and exception handling. In particular, they record a DWARF program, a pointer to a *personality* routine, as well as a pointer to a so-called *language specific data area* (LSDA) (held in the `.gcc_except_table` section).

The DWARF program (split between the CIE and FDE) specifies how to unwind the stack. To be precise, it is a branch-free sequence of instructions that describe how to build a table that can be used to unwind the stack. These instructions are labeled $DP_{\#}$ in the figure. The table contains a row for each address in the FDE's range, and a column for each register in the program. An entry in the table, when defined, is an offset from the *canonical frame address* (CFA) where a register is stored during an unwind event. The CFA is typically represented by the stack pointer, and this fact is typically specified by the DWARF program in the CIE.

Where the DWARF program specifies how to unwind the stack, the personality routine and LSDA help the runtime system call destructors and catch exceptions. The personality routine knows how to parse the LSDA. Technically, the LSDA is language- and compiler-specific, but every language and compiler we examined use the same format. All programs using the same format is a direct consequence of how GCC implements the exception handling in its language-agnostic backend and other compilers striving for binary compatibility with GCC.

Besides additional fields to support denser encoding, the LSDA has a table with an entry for each call site (CS) in the FDE's range. An entry in the call site table encodes the address of the call site, as well as information about what types of exceptions should be caught, and if the call site has a pointer to code (called a *landing pad* or LP) that performs the necessary cleanup. In the figure, `func3` has a string that is allocated on the stack, and if the call to `func1` throws an exception, that string must have its destructor called. Consequently, an entry in the LSDA specifies where the call site is, that actions needs to be taken on an exception, where the code is to perform those actions (the LP), and further specifies that the call site catches exceptions of type int.

The `.eh_frame_hdr` section is optional and used to help the runtime locate the proper entries in `.eh_frame` more quickly by using a binary search instead of a linear search. The `.eh_frame`

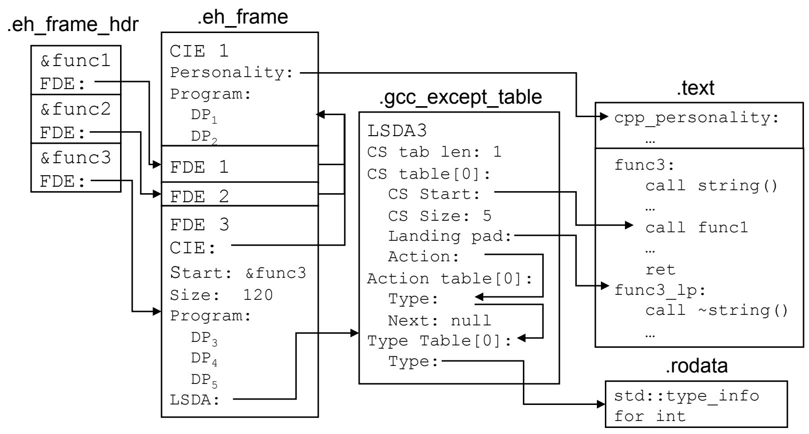

Figure 2: A simplified, high-level example of the exception handling information stored in an ELF binary.

section cannot be binary searched directly because of the variable length encodings of fields.

Together, these individual tables and landing pads are used to unwind the stack and invoke the proper destructors. Tables are generally indexed based on return address but may be indexed by other program counters in the event of a forced unwinding, for example from a signal handler or call to `pthread_exit()`. Additional detail can be found in other publications [1, 12].

Because of their dense, range-based encoding, it is difficult to do small edits to these tables. For example, relocating a call instruction to a new area requires that an FDE be split to accommodate the new memory area, and the call site address in the call site table be updated. But since the FDE, CS, action, and type table fields are variable-length encoded and relative to the FDE's starting address, it is likely that the new call site table entry will have fields that are longer or shorter than the previous entry. Changing a field size requires shifting subsequent fields, which in turn forces updating more offsets. A cascade of edits can occur, causing the entire table to change.

Because a rewriter's goal is to easily make adjustments to a program, compose transformations, and create an output file that is still efficient, we eschew an IR that directly deals with all these complexities. Instead, we flatten the data structures and remove all the fields used for space efficiency. Instead of using variable length encodings, we used 32-bit fixed length encodings for most fields. In particular, for each machine instruction in the program we are rewriting, we record:

- The exact DWARF program for unwinding the stack (combining the CIE and FDE portions), stored as an array of DWARF instructions. Since the CIE and FDE DWARF program is used for all the machine instructions in the FDE's range, some DWARF instructions are unnecessary for unwinding at some machine instruction. We trim DWARF programs accordingly.
- The instruction in the IR that represents the start of the personality routine.
- If the instruction is in the call site table, we record the landing pad, and relevant action and type table information necessary to invoke the personality routine properly.

Because we were unable to find an existing, suitable library to parse the ELF `eh_frame` sections, we built our own parser for it that builds an abstract syntax tree style (AST) representation. We use the AST to populate the initial IR with the aforementioned information. Since many machine instructions typically share the same trimmed DWARF program, the IR provides copy-on-write and automatic de-duplication facilities to shield a transform writer from the complexities of managing these fields.

2.3 EH Frame IR Manipulation

If a transformation is to be enacted by the binary rewriter, the unwind and exception information must be updated. For example, if the stack size is extended for security purposes (say, to store a canary value), the unwind information must be updated accordingly, otherwise the stack unwinder is going to give erroneous results if an exception is thrown, likely resulting in a program crash.

Because we elide the complexities of directly encoding the ELF tables in our IR, it is quite easy for a transform writer to update the EH information. For example, if the transform writer wants to put additional checks when catching an exception, it is easy for them to find and adjust the landing pad to include new code. If they want to add a catch all clause to a particular call site, they can simply add the appropriate entry directly to the call's action and type table. Similarly, inserting new instructions can easily support exception handling and unwinding by copying the EH information from an adjacent instruction. If the newly inserted instructions modify the stack (e.g., saves and restores a register), the DWARF program for unwinding can be easily extended to restore the register.

Perhaps the most complicated case is when a transform decides to change a function's frame layout. In this case, the EH information for every instruction in the program is likely to change. A transform writer would have to iterate through all the function's instructions, and then update the DWARF program for each to reflect how the stack has changed. Section 3.2 discusses a sample transform that performs exactly these edits. The EH frame editing took about 250 additional lines of code to implement, much of it very simple code to iterate the function's instructions and edit the corresponding DWARF programs.

2.4 EH Frame IR Instantiation

After the constructed IR has been manipulated to provide the user-selected enhancements, a new ELF file must be constructed before we can execute the program. As such, the IR must be mapped back into the ELF `eh_frame` format. Since the information contained in the tables is based on ranges of return addresses, we first layout the code so that all instructions are assigned to final addresses in the program.

Next, we construct an `eh_frame`. To achieve this goal, we first create an abstract representation of each structure in memory.

We could create a new abstract FDE, CIE and LSDA for each instruction in the output program, but the size overhead would be undesirable. Instead, we start with an empty CIE and FDE and create an LSDA for each FDE. Then we iterate through each instruction in the program starting at low addresses and working towards high addresses. At each instruction, we determine if the most recent CIE and FDE can be extended to cover the instruction. To qualify as being extendable, the CIE's DWARF program must match, and the FDE's DWARF program must be a prefix of the under-consideration instruction's program, and the personality routine must match. If the structures can be extended, they are. New call site table entries, action table entries, and type table entries are added to the FDE's LSDA as appropriate. If the FDE or CIE structures cannot be extended, a new abstract FDE or CIE is created and added to the list.

Once the abstract representation is constructed, we concretize it by emitting the abstract representation to a file as assembly code and use the system assembler to create the encoded, compressed binary representation. We use assembler labels for linking the structures within the section, and absolute values for any code addresses (that were previously assigned during code layout). The binary form is extracted from the assembler's output file, and added directly to final ELF file.

This method has the benefit of re-using the same mechanism the compiler uses for creating the unwind and EH tables, while allowing the code layout algorithm to be selected by the rewriter. The downside is that the code layout can dramatically affect the size of the rewritten EH tables. In Section 3.1, we discuss the impact of code layout on EH table size.

3 EXPERIMENTAL RESULTS

3.1 Performance and Filesize Evaluation

To measure the effectiveness of our EH-table rewriting, we performed a number of experiments using the SPEC CPU2006 Benchmark suite [11]. The experiments were performed on an Intel 2.4GHz E4645 processor (12 cores) with 48Gb of main memory running Ubuntu 14.04.2 LTS. The binaries were produced using gcc, g++, gfortran version 4.8.4 with -O2 optimization level (with the exception of perl and wrf, which require lower optimization levels for the program to operate correctly before we apply any rewriting). The benchmark DealII was not included because it does not build correctly at any optimization level.

We measured the performance (runtime and disk file size) of four configurations: 1) blocks of code are placed randomly, 2) blocks are placed to improve locality, 3) random placement with EH frame rewriting, and 4) optimized placement with EH frame rewriting. In Figures 3, 4, and 5 performance results are normalized to the performance of the original binary (i.e., less than 1.0 indicates speedup/space decrease and greater than 1.0 indicates slowdown/size increase).

Figure 3 shows the runtime overhead of the four configurations. The difference between bars 1 and 3 shows that EH-table rewriting significantly improves performance (improves geometric mean from 1.15 to 1.07). The improvement from EH-table rewriting comes from two sources. First, because return addresses are not pinned (Section 2.1), the rewriter can leverage the hardware's `call` instruction to write the return address instead of having to emit a sequence of instructions to write the return address into the proper place. Using the hardware's call instruction further improves performance because the hardware's branch predictor expects calls and returns to be matched.

Second, because the new EH tables reflect the actual location of return address targets in the rewritten code, these addresses no longer need to be pinned giving greater flexibility to placement of blocks of rewritten code. In the figure, we include bars 1, 2 and 3 to highlight the additional benefits of giving the code placement algorithm more flexibility. Bars 1 and 2 show the benefit of using a locality based code placement algorithm—an improvement of the geometric mean from 1.15 to 1.11. The difference of the last two bars show the benefit of locality when EH-table rewriting unpins return addresses — the performance average drops from 1.07 to 1.03, just 3% above the original executable.

As noted in Section 2.4, the rewriting of the binary affects EH-table size and thus the size of the binary. Figure 4 presents comparisons of the file size of the binaries. The first two bars show the base effect of Zipr on file size. This overhead comes from two sources: fragmentation due to pinning and the longer calling sequence when setting the return address to a pinned value. The last two bars shows the effect of EH-table rewriting on size. Because Zipr's code layout algorithms are agnostic to EH table size considerations, a much larger number of FDEs are created over the baseline. Our measurements (not shown) indicated that with no locality optimization there were on average 4.7 times more FDEs than in the baseline configuration. The random placement results in more FDEs because an FDE is inherently range based. By splitting up code blocks covered by one FDE in the original, more FDEs are required.

With locality optimization, the average number of FDEs dropped to 3.2 times more FDEs than in the baseline configuration. This drop is because the locality layout attempts to keep blocks from the same function together. This inadvertently lowers the FDE count.

While file size is a secondary consideration for some application areas, it can be important and optimizing FDE creation is an area of future work. In the future, an FDE-aware layout algorithm can likely make significant gains in reducing file size.

3.2 Case Study: Stack Layout Transformation

In the previous section we reported measurements of the performance of simply rewriting a binary—in essence a null transform. In the null transform it is not mandatory to modify the EH tables (i.e., bars 1 and 2 in Figures 3 and 4). However, when new functionality (e.g., a security enhancement) is added to an application, it is necessary to update the EH tables. As a preliminary investigation of

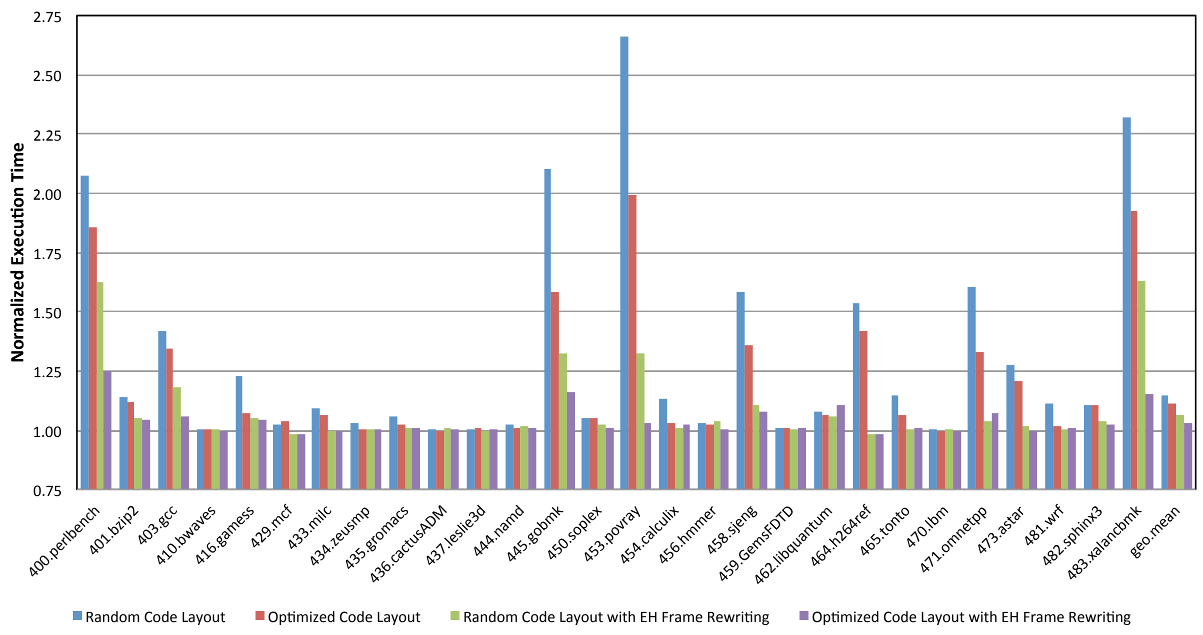

Figure 3: Performance Overhead of SPEC CPU2006 Benchmarks (normalized to native execution).

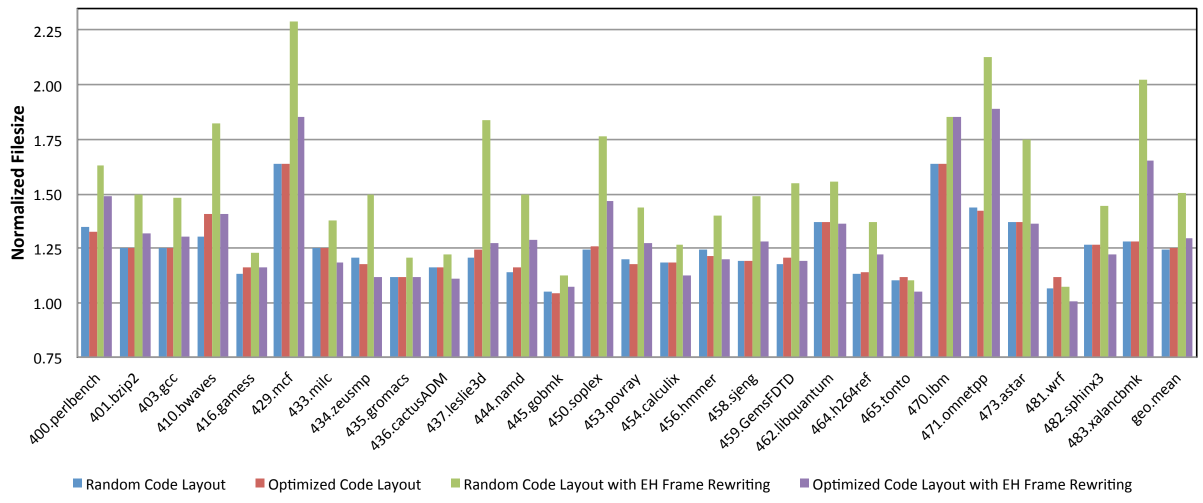

Figure 4: Filesize Overhead of SPEC CPU2006 Benchmarks.

composing security techniques with exception handling, we applied a diversity transform to the six working C++ benchmarks in SPEC CPU2006. We choose the C++ benchmarks because they are often omitted from rewriter evaluations because they are particularly tricky to rewrite due to the exception handling and large code size.

For these benchmarks, we applied a stack-layout diversity transform (SLX), that randomly pads the activation records of functions and inserts canaries [8]. This transform has proved effective at preventing certain types of stack buffer overflow attacks.

Figure 5 shows the performance overhead of two configurations when SLX is applied. For bar 1, EH-table rewriting is not applied, and for bar 2 EH-table rewriting is enabled.

The first thing to notice is that bar 1 is missing for `453.povray` and `471.omnetpp`. The missing bars are because these applications dynamically throw exceptions, which causes the benchmark to fail unless the EH tables are updated. These results highlight the need for a robust binary rewriter to handle applications that throw exceptions. The geometric mean for the working benchmarks in

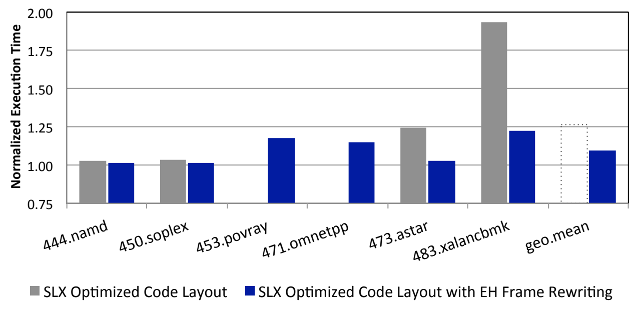

Figure 5: Normalized Performance Overhead of SLX Transform on SPEC CPU2006 C++ Benchmarks.

this configuration is 1.26 (displayed as the the 'ghost' bar). For SLX with EH-table rewriting (bar 2), the geometric mean is 1.1.

3.3 Case Study: Webservers, Ada and Libraries

To demonstrate that our EH-table rewriting is robust under a wider range of EH handling situations, we decided to try an Ada program. Ada is different from C++ because Ada uses exception handling as part of normal operation. For example, the end-of-file (EOF) conditions are typically checked by trying to read a byte, and an exception is thrown/caught to indicate EOF.

We looked for a large, open-source project written in Ada, and found the Ada Web Server (AWS) [2]. AWS is an Ada library that provides web services for other programs. As part of the distribution, it comes with samples for how to use it, and we chose the WPS (web page server) example as it provides basic web page services that are easy to test. We noted that a single web page request often saw tens or even hundreds of exceptions thrown, far more than all of SPEC executes during an entire run of the test suite. Thus, we believe AWS to be a suitable test of the functional correctness of our EH-table rewriting.

We compiled WPS with the GNU `gnat` compiler using optimization level `-O2`. We then performed a rewrite with and without EH-table rewriting on the main executable (WPS statically links the AWS code into the main executable), as well as two Ada runtime libraries: `libgnat.so` and `libgnarl.so`. The libraries are written mostly in Ada with some C, C++, and assembly. They support common Ada operations like file manipulation, I/O, etc., much like `libc.so` or `libstdc++` for C and C++. The three files total 11 megabytes of disk space and make for a substantial test.

After rewriting, we used Apache Jmeter to run a battery of web requests. We observed no failures and conclude that the rewriting is robust for Ada as well as for system libraries.

For the three rewritten executables, we observed file size overheads of 94.8%, 44.3%, and 56.7% after EH-table rewriting (with the optimized code layout). Unfortunately, we were unable to adequately measure performance overheads, as our test setup made the machine I/O bound, and presenting performance numbers would be misleading.

4 RELATED WORK

There are many static binary rewriters (*e.g.*, [20], [10]). Some are designed to transform the input binary to accomplish a particular task (*e.g.*, [19], [18]). Some work only at linking phase ([13]) or require the programs' debugging symbols and relocation information (e.g., [7]). Some do not provide the transformation writer with a high-level API for developing transformations ([9]).

Recent notable static binary rewriting platforms that do not require debugging information or other metadata include SecondWrite [3], UROBOROS [17], and Ramblr [15].

SecondWrite recreates an IR from the input binary, applies user-specified transformations to that IR and, finally, passes the transformed IR to the LLVM compiler to generate the rewritten program. SecondWrite splits the original program stack into individual frames, splits those frames into individual variables and, finally, converts constants and variable memory accesses into symbols. Based on that analysis, SecondWrite constructs an IR from the original program that can be analyzed by LLVM. SecondWrite applies LLVM's built-in optimizations to this IR and uses LLVM's code generation algorithms to reconstruct the modified program.

UROBOROS and Ramblr recreate reassembleable disassembly [16] from the input binary, apply user-specified transformations to that representation and, finally, pass the modified version of that representation to an assembler to generate the rewritten binary. Reassembleable disassembly is different than the disassembly output from a tool like objdump or IDA Pro which contains constant immediates to address other code or data using their locations in the input binary. Because absolute addresses are used as pointers, the instructions and data at those addresses cannot be moved and, therefore, the output from a traditional disassembler cannot be given to an unmodified assembler to recreate the program. In reassembleable disassembly, those code/data immediate values have been replaced with symbols which gives an unmodified assembler the ability to conveniently and arbitrarily place data and code and update the symbolic addresses with the absolute addresses as the last step before completing the rewriting process.

To the best of our knowledge, SecondWrite, UROBOROS, and Ramblr have left exception handling support for future work.

5 SUMMARY

Static binary rewriting is emerging as an important tool for late stage-modification of binaries. To be widely adopted, binary rewriters must be robust and widely applicable. This paper has described and evaluated an approach for rewriting applications that make use of stack unwinding for exception handling. Beyond expanding the use of static binary rewriting to an important class of applications, the technique also improves performance. Measurements using the full SPEC CPU2016 benchmark suite showed that when the technique was incorporated into a modern, fully-featured binary writer, runtime overhead was reduced from 15% to 3%.

ACKNOWLEDGMENTS

The authors thank the anonymous referees for their valuable comments and helpful suggestions. The work is supported by AFRL and DARPA under Grant Nos.: FA8750-15-C-0118, FA8750-14-C-0110 and FA8750-17-2-0079.

REFERENCES

[1] [n. d.]. The DWARF Debugging Standard. ([n. d.]). http://www.dwarfstd.org

[2] AdaCore. 2017. AWS Ada Web Server. (2017). http://libre.adacore.com/tools/aws/

[3] Kapil Anand, Matthew Smithson, Khaled Elwazeer, Aparna Kotha, Jim Gruen, Nathan Giles, and Rajeev Barua. 2013. A compiler-level intermediate representation based binary analysis and rewriting system. In *Proceedings of the 8th ACM European Conference on Computer Systems - EuroSys '13*. ACM Press, New York, New York, USA, 295. https://doi.org/10.1145/2465351.2465380

[4] Mahmoud Elnaggar, Jason Hiser, Tony Lin, Anh Nguyen-Tuong, Michele Co, Jack Davidson, and Nicola Bezzo. 2017. Online Control Adaptation for Safe and Secure Autonomous Vehicle Operations. In *NASA/ESA Conference on Adaptive Hardware and Systems*.

[5] William H. Hawkins, Michele Co, Jason D. Hiser, Anh Nguyen-Tuong, and Jack W. Davidson. 2017. Zipr: Efficient Static Binary Rewriting for Security. In *The 47th IEEE/IFIP International Conference on Dependable Systems and Networks*.

[6] William H. Hawkins, Jason D. Hiser, and Jack W. Davidson. 2016. Dynamic Canary Randomization for Improved Software Security. In *Proceedings of the 11th Annual Cyber and Information Security Research Conference on - CISRC '16*. ACM Press, New York, New York, USA, 1–7. https://doi.org/10.1145/2897795.2897803

[7] Michael A. Laurenzano, Mustafa M. Tikir, Laura Carrington, and Allan Snavely. 2010. PEBIL: Efficient static binary instrumentation for Linux. In *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*. IEEE, 175–183. https://doi.org/10.1109/ISPASS.2010.5452024

[8] Benjamin D. Rodes, Anh Nguyen-Tuong, Jason D. Hiser, John C. Knight, Michele Co, and Jack W. Davidson. 2013. *Defense against Stack-Based Attacks Using Speculative Stack Layout Transformation*. Springer Berlin Heidelberg, Berlin, Heidelberg, 308–313. https://doi.org/10.1007/978-3-642-35632-2_29

[9] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian Bershad, and Brad Chen. 1997. Instrumentation and Optimization of Win32/Intel Executables Using Etch. In *Proceedings of the USENIX Windows NT Workshop (NT'97)*. USENIX Association, Berkeley, CA, USA, 1–1. http://dl.acm.org/citation.cfm?id=1267658.1267659

[10] Amitabh Srivastava, Alan Eustace, Amitabh Srivastava, and Alan Eustace. 1994. ATOM: A System for Building Customized Program Analysis Tools. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation - PLDI '94*, Vol. 29. ACM Press, New York, New York, USA, 196–205. https://doi.org/10.1145/178243.178260

[11] Standard Performance Evaluation Corporation. 2006. SPEC CPU2006 Benchmarks. (2006). http://www.spec.org/osg/cpu2006.

[12] Ian Lance Taylor. 2011. Airs - Ian Lance Taylor. (2011). http://www.airs.com/blog/archives/460

[13] L. Van Put, D. Chanet, B. De Bus, B. De Sutter, and K. De Bosschere. 2005. DIABLO: a reliable, retargetable and extensible link-time rewriting framework. In *Proceedings of the Fifth IEEE International Symposium on Signal Processing and Information Technology, 2005*. IEEE, 7–12. https://doi.org/10.1109/ISSPIT.2005.1577061

[14] Mike Walker. 2015. Machine vs. Machine: Lessons from the First Year of Cyber Grand Challenge | USENIX. (2015). https://www.usenix.org/node/190798

[15] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. [n. d.]. Ramblr: Making Reassembly Great Again. ([n. d.]). https://doi.org/10.14722/ndss.2017.23225

[16] Shuai Wang, Pei Wang, and Dinghao Wu. 2015. Reassembleable Disassembling. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Washington, D.C., 627–642. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/wang-shuai

[17] Shuai Wang, Pei Wang, and Dinghao Wu. 2016. UROBOROS: Instrumenting Stripped Binaries with Static Reassembling. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 236–247. https://doi.org/10.1109/SANER.2016.106

[18] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. 2012. Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*. ACM, New York, NY, USA, 157–168. https://doi.org/10.1145/2382196.2382216

[19] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. 2012. Securing Untrusted Code via Compiler-agnostic Binary Rewriting. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC '12)*. ACM, New York, NY, USA, 299–308. https://doi.org/10.1145/2420950.2420995

[20] Mingwei Zhang, Rui Qiao, Niranjan Hasabnis, R. Sekar, Mingwei Zhang, Rui Qiao, Niranjan Hasabnis, and R. Sekar. 2014. A platform for secure static binary instrumentation. *ACM SIGPLAN Notices* 49, 7 (sep 2014), 129–140. https://doi.org/10.1145/2674025.2576208