Incremental CFG Patching for Binary Rewriting
Extended Abstract

Xiaozhu Meng[*1] and Weijie Liu[†2]

[1]*Department of Computer Science, Rice University*
[2]*Luddy School of Informatics, Computing, and Engineering, Indiana University Bloomington*

1. Motivation

Binary rewriting instruments compiled executables and libraries without their source code and has significant application to software security [5, 18, 11, 12], software correctness [8]. and performance analysis [17, 14, 15]. Current binary rewriting techniques either rely on several limiting assumptions on binaries to achieve complete binary analysis to perform IR lifting, or patch individual instructions without utilizing any binary analysis, which leads to prohibitive runtime overhead.

In this paper, we design a new general binary rewriting approaching, *incremental CFG patching*, to balance the runtime overhead and binary rewriting generality. Our approach supports multiple architectures, including `x86-64`, `ppc64le`, and `aarch64`and multiple programming languages, including C/C++, Fortran, Rust and Go.

2. Limitations of the State of the Art

A rich literature of binary rewriting research is devoted to improving the runtime overhead, reliability, and scalability of binary rewriting [13, 2, 6, 16, 7].

Recent approaches for binary rewriting have taken two opposite directions. On one hand, researchers utilize meta-data available in binaries to perform complete binary analysis to lift binaries to IR and then re-generate new ones; we call this approach as *IR lowering*. Egalito [16] and RetroWrite [6] are two examples in this category, leveraging relocation information in Position Independent Executable (PIE) and achieving binary rewriting with near to zero overhead in their empirical evaluation.

However, this IR lowering approach has two major disadvantages. First, complete analysis is an undecidable problem in general and is difficult to achieve in many practical use cases. Tools based on IR lowering do not support source language specific features such as C++ exceptions and `.vtab` function tables in Go binaries even when these programs are compiled into PIE. In other words, PIE does not necessarily make full binary analysis easy. In addition, while PIE is the future trend, position dependent code cannot be ignored. Current supercomputers and servers typically run Red Hat 7 systems, on which PIE is not the default. Red Hat 7's maintenance support is scheduled to end in 2024 [10]. Even on Linux distributions whose default GCC compilers emit PIE by default, vendor specific compilers may make a different decision: on Intel Dev Cloud, we have Ubuntu 18.04, whose system GCC compiler will emit PIE; but the Intel toolchain on that system will emit position dependent code.

Second, IR lowering presents an "all-or-nothing" dilemma to its users. As it must lift all binary functions to IR, if one of the functions in binary contains rare, difficult binary code construct, the whole binary rewriting may fail. IR lowering by design does not allow leaving certain functions untouched while rewriting the other ones. This "all-or-nothing" tradeoff is reasonable for security applications such as software hardening [5, 12, 11]. It would not generate a partially hardened binary that brings a false sense of security. However, this tradeoff may not be ideal for other application domains. For example, for performance analysis, the users may have known that certain functions are not the bottleneck, and want to focus on a subset of the functions in the binary.

On the opposite to IR lowering is *instruction patching*, which does not use any binary analysis, to achieve reliable binary rewriting. E9Patch [7] devise multiple instruction sequences as trampolines to transfer control flow from original code to instrumentation. This approach has the advantage of being able to handle source language specific features and allowing partial binary rewriting. However, it incurs prohibitive runtime overhead as every instrumented instruction will require a branch from original code to instrumentation and a branch back to original code: it incurs over 100% runtime overhead when instrumenting basic blocks with empty instrumentation. In addition, this approach does not guarantee high level instrumentation semantics. For example, instruction patching cannot ensure the semantics of function entry instrumentation, which should be executed once and only once when a function is called. If the function entry address is inside a loop, without constructing the CFG and modifying the back edge of the loop, the function entry instrumentation will be executed per loop iteration [2].

3. Key Insights

The basic idea of our approach is to use trampolines to catch control flow that we cannot accurately rewrite, which aims for generality and partial instrumentation, and use binary analysis to identify the necessary places to install trampolines and rewrite as much control flow as possible, which helps reduce runtime overhead.

The foundation of our approach is a *trampoline placement analysis*. We define *control flow landing (CFL) blocks* as the basic block where control flow can be transferred from instrumentation back to the original code, and establish that it

[*] xm13@rice.edu
[†] weijliu@iu.edu

is sufficient to install trampolines at only CFL blocks.

A key reason why code patching incurs high overhead is due to the control flow bouncing between the original code and the rewritten code. If we can reduce the number of CFL blocks, we can then reduce this control flow bouncing, and thus reduce runtime overhead. This insight guides us to remove as many CFL blocks as possible. Two categories of CFL blocks are (1) jump table target blocks, which can be removed if we rewrite jump tables so that intra-procedural indirect jumps would stay in the rewritten code, and (2) function entry blocks, which can be removed if we rewrite function pointers.

To handle language specific features such as C++ exceptions and stack unwinding in Go's runtime used for memory garbage collection and dynamic stack growing, an existing approach is to emulate a call instruction with a 3-instruction sequence [3, 1], which puts the return address of the original call instruction to the stack. In this way, stack unwinding can be performed normally. However, this requires emulating every function call and we observe over 30% of runtime overhead by just emulating function calls, and also cause call fall-through blocks to be CFL blocks as the original return addresses will be pushed to the stack.

Inspired by dynamic binary translation used for dynamic instrumentation, we use a runtime routine that translates the return address from the rewritten code to the corresponding original call site before the return address is used for unwinding. In this way, we no longer need to emulate function calls and call fall-through blocks are no longer CFL blocks.

4. Main Artifacts

First, we design a new static analysis, *Trampoline Placement Analysis*, which places trampolines at carefully selected locations to reduce the use of trap based trampolines. The analysis tolerates over-approximated control flow, which may lead to higher runtime overhead, and fails in a safe way for functions that have under-approximated control flow.

Second, we provide three binary rewriting modes that rewrite (1) direct control flow, (2) intra-procedural indirect control flow, and (3) inter-procedural indirect control flow. We characterize the assumptions made by binary analysis used to rewrite these types of control flow and assess their impacts on binary rewriting when the assumptions are violated. This assessment leads to several improvements for rewriting indirect control flow, and gives users an understanding of choices for binary rewriting, avoiding the "all-or-nothing" scenario.

Third, we design *Runtime Return Address (RA) Translation* to translate the return address from the rewritten code to the corresponding original call site before the return address is used for unwinding. This technique enables low overhead binary rewriting for C++ exceptions and Go binaries.

Fourth, we design new trampoline instruction sequences that have varied branching ranges and lengths to further avoid trap based trampolines. All our new trampoline sequences are position independent, which ensures that our techniques work with shared libraries and PIEs. Existing work focuses designing trampoline instruction sequences for `x86-64` [4, 7]. We learn from existing work and also design new trampolines for `ppc64le`and `aarch64`.

We implement incremental CFG patching as an extension to the Dyninst binary analysis and instrumentation tool suite [9] and we will work with Dyninst developers to upstream our work.

5. Key Results and Contributions

We evaluated our approaches with SPEC 2017 CPU, Firefox's `libxul.so` (includes code written in C, C++, and Rust), and Docker (written in Go). We can successfully rewrite over 99.41% of the total functions in binaries from SPEC CPU 2017, 99.93% functions in `libxul.so` and all functions in Docker. The average runtime overhead incurred by our approach is under 1% for SPEC CPU 2017 and 2% for Firefox.

We present a case study with Diogense [14, 15], which is a tool for automatically identifying and fixing unnecessary CPU/GPU synchronization and duplicated CPU/GPU memory transfers. Diogenes has a step that uses Dyninst to instrument Nvidia runtime driver `libcuda.so` to identify the hidden synchronization function in the driver. We speed up the identification from 30 minutes to 30 seconds by replacing mainstream Dyninst with our implementation.

In summary, this work makes the following contributions:

- Incremental CFG patching, a general binary rewriting approach that balances runtime overhead and generality, which supports three architectures and five source programming languages;
- Trampoline placement analysis that reduces trap-based trampolines and tolerates control flow over-approximation;
- An assessment of imprecision in binary analysis for rewriting indirect control flow and improvement to binary analysis for binary rewriting
- Runtime RA translation, an efficient mechanism to rewrite stack unwinding, which is necessary for programs that use C++ exceptions and programs written in Go;
- An implementation of our new techniques in Dyninst and a case study illustrating how our new techniques speed up an existing software tool.

6. Why ASPLOS

This paper describes a new binary rewriting approach, which supports multiple architectures and programming languages. Our approach utilizes specific features of the ISAs to design position independent trampolines, and efficiently address language specific features such as C++ exceptions and Go's builtin stack unwinding. While we attempt to avoid trap based trampolines, the interaction with operating systems is still necessary as a last resort. We also show case the value of our approach with an instrumentation based tool that analyzes CPU/GPU synchronization.

References

[1] Erick Bauman, Zhiqiang Lin, and Kevin W Hamlen. Superset disassembly: Statically rewriting x86 binaries without heuristics. In *Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, Feb. 2018.

[2] Andrew R. Bernat and Barton P. Miller. Structured binary editing with a cfg transformation algebra. In *2012 19th Working Conference on Reverse Engineering (WCRE)*, page 9–18, Kingston, ON, Canada, Oct. 2012.

[3] Andrew R. Bernat, Kevin A. Roundy, and Barton P. Miller. Efficient, sensitivity resistant binary instrumentation. In *The International Symposium on Software Testing and Analysis (ISSTA)*, Toronto, Canada, July 2011.

[4] Buddhika Chamith, Bo Joel Svensson, Luke Dalessandro, and Ryan R Newton. Instruction punning: Lightweight instrumentation for x86-64. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 320–332, 2017.

[5] Thurston HY Dang, Petros Maniatis, and David Wagner. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 555–566. ACM, 2015.

[6] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *41st IEEE Symposium on Security and Privacy (Oakland)*, May 2020.

[7] Gregory J. Duck, Xiang Gao, and Abhik Roychoudhury. Binary rewriting without control flow recovery. In *41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, London, UK, June 2020.

[8] Yizi Gu and John Mellor-Crummey. Dynamic data race detection for openmp programs. In *International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, Dallas, Texas, Nov. 2018.

[9] Paradyn Project. Dyninst: Putting the Performance in High Performance Computing, http://www.dyninst.org.

[10] Red Hat. Product Life Cycles, `https://https://access.redhat.com/product-life-cycles?product=Red%20Hat%20Enterprise%20Linux`, accessed Aug. 12, 2020.

[11] V. v. d. Veen, E. Göktas, M. Contag, A. Pawoloski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *2016 IEEE Symposium on Security and Privacy (SP)*, San Jose, CA, USA, May 2016.

[12] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical context-sensitive cfi. In *22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, Denver, Colorado, USA, 2015.

[13] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. Ramblr: Making reassembly great again. In *24th Annual Symposium on Network and Distributed System Security (NDSS)*, San Diego, CA, USA, Feb. 2017.

[14] Benjamin Welton and Barton P. Miller. Diogenes: Looking for an honest cpu/gpu performance measurement tool. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, 2019.

[15] Benjamin Welton and Barton P. Miller. Identifying and (automatically) remedying performance problems in cpu/gpu applications. In *34th ACM International Conference on Supercomputing (ICS)*, Barcelona, Spain, June 2020.

[16] 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 *Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Lausanne, Switzerland, March 2020.

[17] David Williams-King and Junfeng Yang. Codemason: Binary-level profile-guided optimization. In *3rd ACM Workshop on Forming an Ecosystem Around Software Transformation*, FEAST'19, Nov. 2019.

[18] Mingwei Zhang and R. Sekar. Control flow integrity for cots binaries. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, 2013.