Incremental CFG Patching for Binary Rewriting

Xiaozhu Meng
Rice University
Houston, USA
xm13@rice.edu

Weijie Liu
Indiana University Bloomington
Bloomington, USA
weijliu@iu.edu

ABSTRACT

Binary rewriting has been widely used in software security, software correctness assessment, performance analysis, and debugging. One approach for binary rewriting lifts the binary to IR and then regenerates a new one, which achieves near-to-zero runtime overhead, but relies on several limiting assumptions on binaries to achieve complete binary analysis to perform IR lifting. Another approach patches individual instructions without utilizing any binary analysis, which has great reliability as it does not make assumptions about the binary, but incurs prohibitive runtime overhead.

In this paper, we introduce Incremental CFG Patching, a general binary rewriting approach, to balance the runtime overhead and binary rewriting generality. The basic idea is to utilize code patching to catch control flow that we cannot accurately rewrite and use binary analysis to rewrite as much control flow as possible. A key feature of our approach is that we opportunistically utilize binary analysis and binary meta-data to reduce runtime overhead; but for cases where binary analysis failed or there is no sufficient meta-data to support binary analysis, we can still correctly rewrite the binary with small, additional runtime overhead, or achieve partial instrumentation by skipping certain challenging functions. Our approach supports multiple architectures (`x86-64`, `ppc64le`, and `aarch64`), and multiple source programming languages (C/C++ including C++ exceptions, Fortran, Rust and Go), and works with both position dependent and independent code. The evaluation shows that our new approach on average incurs little runtime overhead with SPEC CPU 2017 (<1%) and small overhead on Firefox (<2%), and can successfully rewrite Docker, which is written in Go. Finally, we present a case study that speeds up an instrumentation based CPU/GPU synchronization analysis tool.

CCS CONCEPTS

• **Software and its engineering** → *Software reverse engineering*; Software performance; • **Theory of computation** → **Program analysis**.

KEYWORDS

binary code patching, trampoline placement, binary analysis reliability

ASPLOS '21, April 19–23, 2021, Virtual, USA

ACM ISBN 978-1-4503-8317-2/21/04…$15.00
https://doi.org/10.1145/3445814.3446765

ACM Reference Format:
Xiaozhu Meng and Weijie Liu. 2021. Incremental CFG Patching for Binary Rewriting. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21), April 19–23, 2021, Virtual, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3445814.3446765

1 INTRODUCTION

Binary rewriting instruments compiled executables and libraries without their source code. It has significant application to software security [12, 25, 26, 32], software correctness [6, 16], and performance analysis [28, 29, 31]. A rich literature of research is devoted to improving the runtime overhead, reliability, and scalability of binary rewriting [7, 14, 15, 27, 30].

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 [30] and RetroWrite [14] 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 [23]. 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 [12, 25, 26]. 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 [15] 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 [7].

In this paper, we fill the gap between these two opposite binary rewriting approaches. We design a general binary rewriting approaching, *incremental CFG patching*, that balances runtime overhead and generality. Our approach supports multiple architectures (`x86-64`, `ppc64le`, and `aarch64`) and source programming language (C/C++ including C++ exceptions, Fortran, Rust, and Go), provides high level instrumentation semantics, supports partial instrumentation and incurs incremental runtime overhead. Here, incremental runtime overhead means that we design several binary rewriting modes where a mode with weaker binary analysis assumptions incurs more overhead.

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. However, several challenges must be addressed to put the use of trampolines and binary analysis in harmony.

First, trap-based trampolines can cause prohibitive overhead. A trampoline must have sufficient branching range to reach the rewritten code and not overrun instructions that will be executed. A last resort is to use a trap instruction to trigger a signal; the signal handler can then perform the transfer. However, when such trap-based trampolines are executed frequently, there can be prohibitive overhead.

Second, binary analysis makes several assumptions about compiler code generation when identifying indirect control flow, including jump tables and function pointers. While binary rewriting builds on binary analysis can reduce runtime overhead when the underlying analysis works, it may generate incorrect rewritten binaries when the assumptions are violated.

Third, existing approaches for supporting stack unwinding in rewritten binaries either brings high runtime overhead or suffers from high engineering complexity. This capability is necessary for rewriting language specific features such as C++ exceptions and Go binaries where Go's runtime natively unwinds the stack for memory garbage collection and dynamic stack growing. One approach is to emulate a call instruction with a 3-instruction sequence [5, 8], 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.

The other approach is to update the meta data used for stack unwinding, such as the `.eh_frame` sections, to reflect the structure of the rewritten binary [21]. `.eh_frame` is encoded in DWARF, which has subtle details with regard to its format encoding. Updating `.eh_frame` requires covering every corner case and catching up with new encoding attributes introduced in DWARF. This leads to high engineering complexity and buggy software.

Incremental CFG patching addresses these challenges with the following new techniques. 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. Trampoline placement analysis relies on the construction of CFG, which is a critical task of binary analysis [13, 20, 24]. We do not rely on accurate CFG construction and present a failure mode Analysis of how failures in CFG construction would impact binary rewriting.

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 stack unwinding. This one level of indirection enables the language runtime for stack unwinding to perform stack unwinding using the original `.eh_frame` section, as if the binary was not rewritten. Our approach enjoys low overhead as we no longer need to emulate function calls and simplicity as we do not need to update complicated DWARF.

Fourth, we design new trampoline instruction sequences that have varied branching ranges and lengths. 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` [11, 15]. 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 [22] and evaluated our approaches with SPEC 2017 CPU, `libxul.so` from Firefox, Docker. 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 [28, 29], 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;
- 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, a mechanism to rewrite stack unwinding without incurring additional overhead, 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.

2 RELATED WORK

In Table 1, we compare existing binary rewriting approaches with our work based on three aspects: (1) what types of control flow are rewritten, (2) how to handle unmodified control flow, and (3) how to support stack unwinding in rewritten binaries. We discuss how these aspects impact binary rewriting runtime overhead and generality. We include BOLT [21] in our discussion even though BOLT is a binary optimizer rather than a general binary rewriting tool.

2.1 Rewriting Control Flow

Intuitive, when more types of control flow are rewritten, the runtime overhead decreases. But to rewrite more control flow, more binary analysis is needed, which may lead to less reliable binary rewriting. We discuss two issues with regard to rewriting control flow. First, what types of control flow are rewritten? We identify three types: rewriting no control flow, only direct control flow, and also indirect control flow. Second, as tools commonly utilize relocation entries for binary analysis, we discuss what types of relocation entries are used, which impacts the generality of binary rewriting. We identify three types: no use of relocation, use of run time relocation, and use of link time relocation,

E9Patch [15] does not rewrite any control flow. It does not use any relocation entries either. This strategy allows E9Patch to achieve great binary rewriting generality, but incurring high runtime overhead.

Multiverse [5] and Sensitive Resistant Binary Instrumentation (SRBI) [7] modify only direct control flow while maintaining original indirect control flow. It is straightforward to modify direct control flow as direct control flow targets are encoded in the branch or call instructions. Neither Multiverse nor SRBI uses relocation entries.

Egalito [30], RetroWrite [14], and our work aim to also rewrite indirect control flow. The two main tasks are rewriting jump tables, which implements switch statements, and rewriting function pointers, which are used for indirect calls. The analysis of jump tables and function pointers often involves multiple instructions, understanding instruction semantic, and memory tracking (i.e. tracking register spills through stack). Egalito and RetroWrite argue that since the default compilers on recent Linux distributions generate PIEs by default, it is possible to achieve complete rewriting of indirect control flow utilizing the necessary run time relocation entries in PIE. Egalito and RetroWrite require run time relocation entries to be available, limiting their use to non-PIE. Our work, on the other hand, aims to rewrite indirect control flow as much as we can, and characterize when it is safe and when it is not safe to do so. We use relocation entries when they are available but do not require them to be present.

BOLT [21] as a binary optimizer requires link time relocations to be able to re-order functions. Link time relocation entries are removed by default. The user must recompile the program by passing flag `-Wl,-q` to instruct the linker to retain link time relocation entries. For the domain of binary optimizer, it is reasonable to assume that the user has source code and can recompile the code as required. Unfortunately, this is typically not a reasonable assumption for other use cases of binary rewriting, including security analysis, software forensics, and reverse engineering.

In summary, our work balances the goal of low overhead by rewriting all types of control flow and achieving generality by not requiring relocation entries.

2.2 Handling of Unmodified Control Flow

Unless a binary rewriting approach can rewrite all control flow, it must ensure correct execution of unmodified control flow. We identify three different categories: no handling of unknown control flow (shown as "NA" in the table), code patching, and dynamic translation (DT).

E9Patch and SRBI use code patching, which install trampolines (typically a branch instruction) to transfer control flow to rewritten code. E9Patch focuses on reducing trap based trampolines on `x86-64`. A 5-byte branch instruction on `x86-64`has branching range ± 2GB relative to the current PC, which is typically sufficient, However, there may not be five bytes available. E9Patch's utilizes redundant instruction prefixes, the 2-byte short branch instruction, and operationally equivalent instructions, to reduce the use of trap based trampolines. E9Patch's technique highly depends on the ISA features of `x86-64`, and cannot be extended to `ppc64le`or `aarch64`. On `ppc64le`and `aarch64`, there will always be space for a branch instruction. However, a branch instruction has only ± 32MB and ± 128MB branching range, respectively, which may not be sufficient when the binaries have large code or data sections. SRBI installed trampolines at every basic block and suffered from trap-based trampolines in several cases based on our empirical evaluation.

Our work also uses code patching for handling unmodified indirect control flow. We design a trampoline placement analysis to determine where trampolines are needed and identify more code bytes that can be safely reused for installing trampolines. We also devise new trampoline sequences to address the issue of branching limit on `ppc64le`and `aarch64`.

Multiverse uses the idea of dynamic translation to handle unmodified control flow for binary rewriting. Dynamic translation is a technique used by dynamic binary instrumentation [10, 18], which modifies an indirect control flow instruction to a function

Table 1: Comparison of binary rewriting approaches. We have two empty entries for BOLT as BOLT's paper [21] does not describe corresponding aspects.

Approach	Rewriting control flow		Unmodified control flow	Stack unwinding
	Types to rewrite	Use of relocation		
BOLT [21]		Link time		Update DWARF
Egalito [30]	Indirect	Run time	NA	NA
E9Patch [15]	No	None	Patching	NA
Multiverse [5]	Direct	None	Dynamic translation	Call emulation
RetroWrite [14]	Indirect	Run time	NA	NA
SRBI [7]	Direct	None	Patching	Call emulation
Our work	Indirect	None	Patching	Dynamic translation

call to a translation function implemented in a runtime monitoring library. The translation function at runtime knows the exact control flow target, lookup whether it is instrumented, and decide whether the control flow should be directed to the original target or instrumentation. Multiverse uses this approach to rewrite indirect control flow and inject the translation function into the rewritten binary. For an indirect control flow transfer, which can be done with one instruction when uninstrumented, dynamic translation requires one dynamic translation function call. This significantly increases runtime overhead.

2.3 Supporting Stack Unwinding

An early approach to support stack unwinding in binary rewriting is call emulation [5, 8]. We emit additional instructions in the rewritten code to push the original return address to the stack on `x86-64`or store the original return address to the link register on `ppc64le`and `aarch64`; we then emit a branch instruction to the actual call target. With call emulation, stack unwinding runtime always sees original return addresses, enabling stack unwinding for rewritten binaries. However, the control flow will return the original call site after the callee returns; it is necessary to handle this type of unmodified control flow with either code patching (installing a trampoline at original call site) or dynamic translation (rewriting every return instruction with a translation call). Multiverse and SRBI use this approach to support stack unwinding. The biggest downside is that it incurs high runtime overhead due to emulation.

A recent approach is based on the observation that language runtime consumes DWARF records in `.eh_frame` to look up unwind recipes and then perform unwinding. Therefore, if we can update the data in `.eh_frame` to reflect the new structure of the rewritten binary, stack unwinding can be supported transparently. BOLT [21] implements this approach. The advantage is that it incurs no runtime overhead. However, this approach brings high software engineering complexity due to subtle encoding formats in DWARF. For example, even though BOLT is widely used in industry settings to optimize performance for large facebook web-services and is a well maintained open source project, BOLT developers still need to deal with bugs introduced by new DWARF encoding for `.eh_frame` [9]. We note that DWARF information generated by even mainstream compilers can be buggy [17], showing that it is not a simple task to generate or rewrite DWARF.

Our work uses the idea of dynamic translation to achieve low overhead support for stack unwinding. In our runtime return address translation, we invoke one return address translation per call frame unwinding. Call frame unwinding is an expensive operation as it looks up DWARF unwinding information and updates register states. Therefore, our return address translation does not bring noticeable runtime overhead. In addition, compared to the approach of updating DWARF, our approach enjoys the simplicity of not dealing with DWARF.

Our approach can also be easily adapted to emerging stack unwinding techniques that are not based on DWARF. `frdwarf` [4] has implemented a new stack unwinding technique that "compiles" `.eh_frame` to machine instructions, so that the language runtime does not have to lookup DWARF unwind recipes each time, but just executable the corresponding machine instructions to unwind the stack. Their evaluation shows that this new stack unwinding technique is 10 times faster than DWARF based unwinding. Updating DWARF is not applicable to such new non-DWARF based unwinding techniques.

3 INCREMENTAL CFG PATCHING OVERVIEW

Figure 1 overviews our approach. Given an input binary shown on the left side, our approach emits a new binary shown on the right side. The rewritten binary has several modified sections and newly added sections. We discuss how these sections are arranged. Similar section arrangement has been used by other binary rewriting approaches [5, 7, 30].

`.text` contains trampolines that transfer the execution to section `.instr`, which contains the relocated code and instrumentation. The analysis of trampoline placement is described in Section 4. It determines where trampolines are necessary.

Unmodified control flow in `.instr` transfers the execution back to `.text`, and a trampoline is often immediately executed to go back to instrumentation. This ping-pong execution between `.instr` and `.text` is a major source of runtime overhead for patching based binary rewriting as it pollutes the instruction cache. We design three modes of binary rewriting: `dir`, which only modifies direct control flow, `jt`, which modifies direct control flow and jump tables, and `func-ptr`, which modifies function pointers in addition to control flow modified by `jt`. In Section 5, we identify the assumptions made by binary analysis for rewriting jump tables and function pointers.

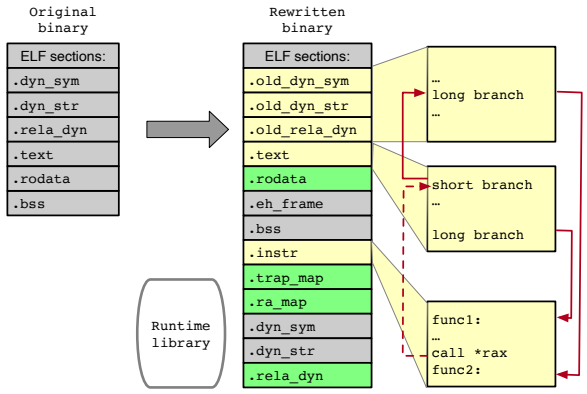

Figure 1: An overview of our approach. Yellow shaded sections contain new code. Green shaded sections contain new data. Solid edges represent control flow caused by trampolines. Dashed edges represent unmodified control flow going from instrumentation back to the original code.

This assessment leads to a jump table rewriting approach that tolerates control flow over-approximation and safely skips detected under-approximated control flow, and a function pointer rewriting approach that requires precise analysis. The requirement of precise function pointers analysis highlights why complete binary analysis is difficult, and it is valuable to have incremental binary rewriting. Rewriting jump tables and function pointers typically modify `.rodata` (read-only data) section and relocation entries in `.rela_dyn`.

Several sections including `.dyn_sym`, `.dyn_str`, and `.rela_dyn` are needed for dynamic linking. These sections are copied and moved to different memory regions so that there will be enough space to hold new dynamic symbols and relocation entries. This is essential to support making function calls to external instrumentation libraries. The old sections are renamed so that the loader will not confuse the old sections with the new ones. We observe that these old sections are no longer used and thus can be safely used as scratch space. Such scratch space in addition to the padding bytes in `.text` is valuable real estate for installing multi-branch trampolines, where we can use a short branch in `.text` to the first branch to scratch space where a long branch is installed. The new trampoline design is described in Section 7.

`.ra_map` (when needed) is a new section added by our approach. We use this section to store a mapping from the return address in `.instr` to the corresponding return address in `.text`, as described in Section 6. This mapping is used at runtime to translate return address on stack to the corresponding original call site, which enables efficient stack unwinding for C++ exceptions and Go binaries. `.eh_frame` is not modified by us.

The last piece is a runtime library, which contains routines to handle trap signals caused by trap trampolines and translate return address on stack. This runtime library can be injected into the rewritten binary at runtime using `LD_PRELOAD`.

4 TRAMPOLINE PLACEMENT ANALYSIS

We present an analysis of trampoline placement that determines where to install a trampoline to ensure safety and reduce the use of trap-based ones. We start with definitions necessary to formulate our analysis, and describe our analysis assuming an accurate CFG. We then discuss the impact of CFG construction failures and imprecision on our analysis.

4.1 Definitions

We use a standard definition for a CFG $G =< B, E, F >$, where

- B is a set of address ranges $[s, e)$ that represents basic blocks. Each of the address range contains at most one control flow instruction at its end, and has incoming control flow only at s;
- E is a set of control flow edges between basic blocks;
- $F \subseteq B$ is a set of function entry blocks.

Control Flow Landing (CFL) Block: a basic block $b \in B$ is called a CFL block if one of its incoming control flow edge is unmodified.

During the execution of the rewritten binary, a CFL block is a block where control flow may be transferred from the rewritten code (`.instr` section) back to the original code (`.text` section). Therefore, we need to install a trampoline to redirect the execution back to the rewritten code before missing any instrumentation.

We use SRBI as an example, which modifies only direct control flow and uses call emulation. For SRBI, CFL blocks include the function entry blocks and indirect jump target blocks, as indirect calls and jumps are not modified. When the binary uses C++ exceptions, CFL blocks also includes call fall-through blocks as the call emulation will push the original return address to the stack, and catch blocks that may catch exceptions.

For a function f, we denote B_{cfl} as the set of CFL blocks in a function and B_{inst} as the set of blocks that are instrumented. We then define:

Instrumentation Integrity: For $\forall b_1 \in B_{cfl} \wedge b_2 \in B_{inst}$, there should be at least one trampoline on every control flow path from b_1 to b_2.

This property specifies that before executing any instrumented basic blocks, a trampoline should be executed to transfer control flow to the rewritten area. Otherwise, instrumentation may be skipped.

We denote B_t as the set of blocks with trampolines installed.

Scratch block: For a block $b \in f$, denote P as a set of blocks that are on any one path between B_t and b, if $P \cap B_{cfl} = \emptyset$, b is a scratch block.

A scratch block will not be executed because all of its reachable control flow paths are intercepted by trampolines. Therefore, scratch blocks can safely be used as scratch space for installing trampolines in B_t.

Trampoline Superblock: For a block $b = [s, e) \in B_t$ in which we need to install a trampoline, we can extend its address range with scratch blocks to $[s, se)$ where $se >= e$ and to a trampoline superblock. Trampoline superblocks create more space to install trampolines and can help to reduce the use of trap-based trampolines.

4.2 Trampoline Placement

Existing approaches install trampolines at every basic block (SRBI), or at every instrumented instruction (instruction patching). These are two sufficient but inflexible strategies to satisfy instrumentation integrity.

We observe that installing trampolines only in CFL blocks also satisfies instrumentation integrity. This strategy has the following advantage over existing approaches: we can incrementally reduce the number of CFL blocks by rewriting more types of control flow, while the instrumented instructions are specified by the user, for which we do not have control. Specifically, we can remove three types of CFL blocks: (1) jump table target blocks if we rewrite jump tables so that intra-procedural indirect jumps would stay in the rewritten code, (2) function entry blocks if we rewrite function pointers, (3) call fall-through blocks with runtime RA translation.

The fewer the CFL blocks, the less likely we will need trap-based trampolines and less runtime control flow bouncing between original code and rewritten code. This analysis bridges the gap between code patching and IR lowering. First, with sufficient binary analysis, code patching can also ensure that program execution stays in the rewritten code, just like rewritten binaries emit by IR lowering. Second, for cases where binary analysis is not sufficient, code patching can utilize trampolines to redirect unmodified control flow, while IR lowering cannot rewrite those binaries.

With the above analysis in place, the algorithm for installing trampolines becomes straightforward. We take as input a function $f = \{b_1 = [s_1, e_1), b_2 = [s_2, e_2), \cdots, b_n = [s_n, e_n)\}$ where $s_1 < s_2 < \cdots < s_n$, and a set of CFL blocks B_{cfl}, the algorithm generates a set of trampoline superblocks where we will install trampolines. The key observation is that every non-CFL block is a scratch block. This is because if the control flow enters the original code, this code block by definition is a CFL block, and thus a trampoline is installed in this block to transfer the execution back to relocated code. Therefore, all non-CFL blocks will never be executed and can be safely used as scratch blocks, and we can extend CFL blocks with these scratch blocks to form trampoline superblocks.

We note that it is possible to design more sophisticated trampoline placement analysis by installing trampolines at blocks that are post-dominated by blocks in B_{cfl} or at blocks that dominate blocks in B_{inst}. This can potentially reduce the number of trampolines even further. Our evaluation in Section 8 shows that our trampoline placement strategies work well in practice.

4.3 Impacts of CFG Construction Failures

To this point, we have assumed that we have an accurate CFG. The construction of CFG from binary code is a critical research area. The focus has been improving the analysis precision of several challenging code constructs, including tail calls, indirect jumps, non-returning functions. While significant research effort has been spent on this topic, to the best of our knowledge, no techniques can guarantee precise CFG construction [3].

In Figure 2, we illustrate three types of failures when constructing CFG and how these different types of failures in CFG construction would impact the quality of binary rewriting. The three types of failures are analysis reporting failure, which means that the binary analysis itself detects errors that it cannot handle, over-approximation, which means the CFG contains infeasible control flow, and under-approximation, which means the CFG misses real control flow.

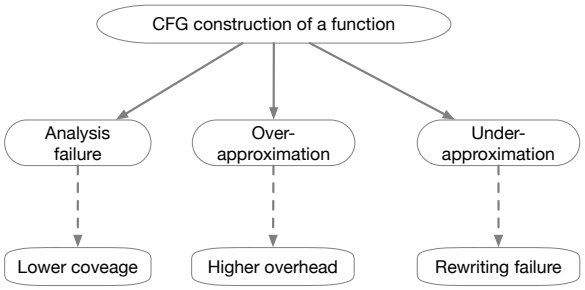

Figure 2: A failure mode analysis of how binary analysis affects binary rewriting. Solid edges represent three possible analysis failures. A dashed edge represent how a particular type of failure of binary analysis would affect binary rewriting.

First, an analysis may fail gracefully when it analyzes a function and report the failure to its user through error handling mechanisms such as error return code or exceptions. Careful software engineering should achieve this goal. For binary rewriting, if CFG construction of a function failed, our approach will simply not instrument this function, which leads to lower instrumentation coverage. We note that the instrumentation of other functions are not affected. To establish this property, suppose the analysis of function f failed and we consider inter-procedure control flow between f and other functions. When f calls an instrumented function, as we always install trampolines at the entry of instrumented functions, there is guaranteed to be a trampoline at the callee's entry, ensuring instrumentation integrity. On the other hand, calls from instrumented functions to f are adjusted to ensure they go to f's entry.

Second, an analysis may report success but over-approximate the CFG. For an over-approximated control flow edge, whose target is address x and falls into block $b = [s, e)$, where $s < x < e$, this over-approximated edge would lead us to create two blocks $b_1 = [s, x)$ and $b_2 = [x, e)$. In the case where we determine b_2 as a CFL block, we would install a trampoline in block b_2, for which we do not actually need. This unnecessary trampoline may reduce scratch space for other trampolines, which may lead to trap-based trampolines. However, it will not lead to wrong instrumentation.

Third, an analysis may report success but under-approximate the CFG. For an under-approximated control flow edge, whose target is address x and falls into a block $b = [s, e)$, where $s < x < e$, we should have two blocks $b_1 = [s, x)$ and $b_2 = [x, e)$, but instead we only see one block $b = [s, e)$. The missed edge is certainly not modified because it is impossible to rewrite control flow without identifying it in the first place. In this case, we will miss a trampoline that should have been installed, which may lead to wrong instrumentation.

In summary, for binary rewriting, we should adapt existing binary analysis to avoid under-approximation failures, as they may lead to catastrophic results.

5 MODIFYING INDIRECT CONTROL FLOW

We have established the safety and performance properties of our approaches. A key factor for reducing runtime overhead is to resolve and rewrite indirect control flow. The computation of indirect control flow can theoretically involve arbitrary instructions and memory locations. Fortunately, mainstream compilers do not do so. Binary analysis for analyzing indirect control flow is based on characterization of compiler code generation patterns. We assess assumptions made by existing binary analysis techniques, discuss the impacts of violating these assumptions on binary rewriting, and design improvements accordingly.

5.1 Intra-procedural Indirect Control Flow

Jump tables represent a typical intra-procedural indirect control flow that implements switch statements. To resolve and rewrite jump tables, we need to identify the following elements: (1) the starting address of a jump table, (2) the number of entries of a table, and (3) the expression of the indirect jump target given a table entry content. These elements are typically identified by performing backward slicing from the indirect jump instruction, computing a symbolic expression of the jump target, and inferring the table size [13, 19, 20, 24].

We design *jump table cloning* where we copy the jump table entries and overwrite several instructions that are used in the jump table computation to reference the new table. The new table ensures the relocated indirect jumps will stay in the relocated area. With this technique, jump table target blocks are no longer CFL blocks.

We retain some information about the resolved jump tables, including table entry size (in bytes), the number of entries, and the symbolic expression that computes the jump target $tar(x)$, where tar represents the function that computes the jump target and x is the content read from a table entry.

Since we know the target addresses of the relocated jump targets, denoted as y, we solve $tar(x) = y$ for x_0 and write the x_0 to the new jump table. After constructing the new jump table, we overwrite the instructions that compute the jump table base to reference the new jump table. On `x86-64`and `ppc64le`, jump table entry has sizes in 4 or 8 bytes, which are sufficient to hold the newly computed table values. On `aarch64`, we find that the compiler often emits jump table entries in 1 or 2 bytes; in this case, we need to modify the jump table read instruction to perform a 4-byte read.

Now we discuss two assumptions made by existing work when resolving jump tables and the impact on binary rewriting when jump table analysis fails.

Assumption 1: Jump table data are not embedded in code.

Egalito [30] made this assumption, which in general holds on `x86-64`and `aarch64`. However, almost all jump tables are embedded within the code section for `ppc64le`binaries, typically placed immediately after the indirect jump that uses the jump table. We do not make this assumption as we perform control flow traversal to identify code and try our best to identify memory accesses to addresses in code sections.

Assumption 2: Jump tables do not overlap with each other and do not overlap with other data.

This is an effective strategy to handle over-approximation of jump tables. Egalito trims jump tables so that they do not overlap. We find that on `aarch64`, jump tables may be separated by other constant data such as strings or numerical values. Our work further trims over-approximation by identifying non-jump table memory accesses and ensure jump tables will not run into other jump tables or known non-jump table data. Note that we do not guarantee that we find all possible non-jump table memory accesses. The more we can find, the better we can reduce over-approximation, but the correctness does not depend on finding all non-jump table memory accesses.

These two assumptions impact the identification of the elements needed to rewrite jump tables. While we expect the analysis of jump tables will evolve and become more robust in the future, we argue that a binary rewriting approach should not rely on precise analysis of jump tables. Following the failure analysis we have presented in Section 4.3, we now discuss how different types of jump table analysis failures will affect binary rewriting, and how we adjust jump table analysis for better binary rewriting. In our experience, these failures are caused by complicated path conditions for tracking values, values spilled to and reloaded from memory, and unhandled instructions that are used in jump table computation.

Failure 1: Cannot find where a jump table starts.

This case corresponds to the "Analysis failure" case shown in Figure 2, which leads to lower instrumentation coverage. We mark such functions uninstrumentable as SRBI [8]. We will show in Section 8 that this strategy may lead to instrumentation coverage lower than 90% of the total functions.

To improve instrumentation coverage, we observed that many failures from jump table analysis are actually caused by indirect tail calls. For indirect tail calls, these jumps do not lead to under-approximated intra-procedural control flow; so, it is safe to instrument the functions that contain indirect tail calls even if we cannot resolve their control flow targets.

A common heuristic to identify indirect tail call is to find whether there are stack frame tear down instructions before the indirect jump [20]. In practice, we find that this heuristic missed many indirect calls for functions that do not need a stack frame. We identify potential indirect tail calls by analyzing the function layout. If an indirect jump is intra-procedural, failing to resolve it would lead to undiscovered code. Essentially, the function would contain gaps in its address range. We decode instructions in the gaps and see if these are real code or just nop padding. If a function contains no gaps or the gaps contain only nop padding, we treat unresolved indirect jumps as tail calls.

We comment that our indirect tail call identification is still based heuristics. Therefore, it is possible that an indirect jump may be wrongly identified as an indirect tail call, when it is actually an intra-procedural indirect jump. Such mistakes would lead to under-approximated CFG, and potentially failures in binary rewriting. As we will shown in Section 8, our new heuristics can empirically improve instrumentation coverage, yet without introducing failures.

Failure 2: Under-approximate the total number of entries in a jump table.

Under-approximated CFG will lead to failures in binary rewriting. We avoid under-approximation based on Assumption 2: we extend the end of the jump table to reach the nearest known data access address or the next jump table start. Note that this may lead to over-approximation, but as we will discuss below, over-approximation does not impact instrumentation safety.

Failure 3: Over-approximate the total number of entries in a jump table.

This can happen when we missed certain non-jump table memory access or failed to find the starts of other jump tables. We clone the jump table to a different location and leave the original jump table unchanged, and the over-approximated entries will not be accessed at runtime. If we overwrite the original jump table in place, the over-approximated entries will cause overwriting to data not in the jump table, and cause the rewritten binary to fail. Therefore, copying the table to a new place is critical to tolerate control flow over-approximation.

5.2 Inter-procedural Indirect Control Flow

Modifying indirect calls and indirect tail calls is a challenging task. The definition of a function pointer can be far away or even in a different function from the use of the function pointer, and the function pointer may flow through multiple memory locations. Therefore, it is difficult to determine whether a value that matches a function entry address is a function pointer or happens to be a numerical value.

To rewrite inter-procedural indirect control flow, we do not need to know where an indirect call may go. We only need to modify the definition of a function pointer. The use of a modified function pointer points to the relocated function accordingly.

We establish a safety requirement for rewriting function pointers: it is safe to modify function pointers only when we can precisely identify all function pointers. If we over-approximate a data value as a function pointer and modifies it, this modified value will not be used for control flow but used in other computations, which will lead to changed program behaviors. If we under-approximate function pointers, then function pointer comparisons may end up with a different result. For example, suppose we have a function f and x is a function pointer that is initialized to point to f. For a condition `x == &f`, if we only modify function pointer at one side, the result of the condition will change.

Egalito characterizes that in PIE, an absolute function pointer needs a relocation entry and a relative function pointer needs a PC-relative instruction, and show that it is possible to achieve precise function pointer identifications for PIEs. We point out that this observation does not necessarily hold.

Listing 1 is a code example from docker (Docker version 19.03.11), which is a Go binary and PIE. At address `0x1666e98`, we load a function pointer, whose content will be modified by the loader as specified by the relocation entry, into `%rcx`. Then `%rcx` is incremented by one and stored into a memory location, which then will be used for an indirect call. If we only modify the relocation entry without realizing the function pointer will be incremented by one, then when we instrument the entry of function `runtime.goexit`, the increment by one will cause the indirect call to call into the middle of instrumentation.

```
000000000168f5e0 <runtime.goexit>:
168f5e0: nop
168f5e1: callq  1669c20 <runtime.goexit1>

0000000001666e40 <runtime.oneNewExtraM>:
...
// Load memory from address 0x3826ab0
1666e98: mov  0x21bfc11(%rip),%rcx
1666e9f: inc  %rcx
1666ea2: mov  %rcx,0x40(%rax)

Relocation table:
3826ab0:  R_X86_64_RELATIVE       168f5e0
```

Listing 1: A function pointer that points to function entry address plus one.

We address this issue by identifying PC-relative values, both based on relocations and PC-relative addressing, and perform forward slicing to track additional computation until the value is stored into a memory location. In this way, we track this irregular this type of indirect call that does not call to function entries specified by function symbols.

In this example, the increment by one essentially skips the `nop` at the function entry of `runtime.goexit`. We are not clear why Go compilers will emit such code. In addition, as this code is in Go's runtime, we observed this example in many Go binaries.

In practice, not every binary contains function pointer comparison and function pointer arithmetics. `func-ptr`mode represents the best case scenario for binary rewriting, which cannot guarantee its correctness in a general case, but empirically works for many real world binaries. If `func-ptr`failed for a binary, the user can use other modes, depending on fewer binary analysis assumptions.

6 RUNTIME RETURN ADDRESS TRANSLATION

To remove the overhead of call emulation and support runtime stack unwinding, we design a runtime translation approach to translate a relocated return address to the original return address before its use.

First, during binary rewriting, we create a mapping for all pairs of relocated return address and original return address, and write this mapping inside the rewritten binary. The runtime library will extract this mapping from the rewritten binary, which is then ready for use during runtime unwinding.

Second, we implement a RA translation routine `RATranslation` inside the runtime library, which takes the current unwinding PC as input and returns the corresponding original PC. Since the recorded return addresses are offsets within the original binary, we need to first adjust the input PC according to the load base address of the rewritten binary, and then perform the mapping lookup. If the input PC is not found, we return the input PC; this case happens naturally when we are unwinding through binaries that are not instrumented.

We then describe how we invoke `RATranslation` to support low overhead C++ exception unwinding and Go binaries. These two

language runtime systems use DWARF based unwinding, which use data stored in `.eh_frame` sections. Our RA translation approach can also work with other non-DWARF based unwinding approaches, such as the optimized stack unwinding method described in frdwarf [4].

6.1 C++ Exception Unwinding

We use function wrapping for C++ exception unwinding. We first compile `libunwind` with exception unwinding support and preload `libunwind` to ensure exception unwinding uses `libunwind` instead of `libgcc_s.so`. We then wrap the `_ULx86_64_step` function from `libunwind` on `x86-64`(`_ULppc64_step` on `ppc64le`and `_ULaarch64_step` on `aarch64`), which unwinds a stack frame given the current register state. We wrap this function in the runtime library to call `RATranslation` This additional translation cost is negligible compared to other stack unwinding operations.

Note that in principle it is feasible to use binary instrumentation to modify the default exception unwinding routines in `libgcc_s.so` to invoke `RATranslation`. This strategy is more general, but requires additional engineering effort to identify which registers and memory locations contain the current input PC.

With this RA translation mechanism, we do not need call emulation and call fall-through blocks are no longer CFL blocks.

6.2 Go Binaries

Unlike C++ exception unwinding where the unwinding code is inside a separate library, the traceback code in Go's runtime is by default linked in the binary. We identify that `runtime.findfunc` and `runtime.pcvalue` are the two functions, which take the unwinding PC as a parameter and perform stack unwinding related tasks. Therefore, we instrument the entries of these two functions with a function call to `RATranslation` and overwriting the input PC of these two functions with the return value from `RATranslation`. Go's ABI specifies that input parameters are passed through stacks, so we identify the stack locations that hold the input PC and overwrite it with the translated return address.

We note that Go allows dynamic linking by passing `-linkshared` flag at compile time. In this case, the traceback code in Go's runtime is in a separate library `libstd.so`; we only need to instrument functions `runtime.findfunc` and `runtime.pcvalue` in `libstd.so`.

In summary, our approach for supporting stack unwinding has the benefits of low runtime overhead and simplicity of not dealing with DWARF, but has the drawbacks of having to tailor instrumentation for different language runtime systems.

7 TRAMPOLINE INSTRUCTION SEQUENCE DESIGN

The techniques described in previous sections significantly reduce the number of CFL blocks and the chances of needing trap-based trampolines. Still, trap trampolines may still be needed when the functions are very small or when the branching range is not sufficient for the instrumentation code region. In this section, we address these issues with a new trampoline design.

Table 2 shows the trampolines instruction sequences. Trampolines for `x86-64`and `aarch64`are pc-relative so they will work on both position dependent and position independent code. On `ppc64le`, the table of content (TOC) register `r2` points to the location of TOC table and the compiler emits position independent code to set the content in `r2` for PIE. Therefore, trampolines using `r2` as the base address on `ppc64le`also support position independent code.

Table 2: Trampoline instruction sequences. The "Range" column shows the ± branching range. The lengths of trampolines are in unit of bytes (B) for `x86-64`and in unit of instructions (I) for `ppc64le`and `aarch64`.

Arch.	Instructions	Range	Len.
`x86-64`	2-byte branch	128B	2B
	5-byte branch	2GB	5B
`ppc64le`	`b`	32MB	1I

	`addis reg, r2, off@high`
	`addi reg, reg off@low`
	`mtspr tar, reg`
	`bctar`
		2GB	4I

`aarch64`	`b`	128MB	1I

	`adrp reg, off@high`
	`add reg, reg off@low`
	`br reg`
		4GB	3I

On `ppc64le`and `aarch64`, the long trampoline needs a scratch register to store the branch target. We use register liveness analysis to find a scratch register. When there is no dead register, on `ppc64le`, we save a register to stack and restore the register after storing the jump target to `tar` register. The `tar` register is a special register reserved for system software. We utilize it for branching. On `aarch64`, if we cannot find a scratch register, we fall back to trap.

On all three architectures, the shorter trampolines in length have shorter branching ranges. In cases we only have space for a short trampoline, which does not provide a sufficient branching range, we design a multi-trampoline approach that we first use a short trampoline to branch to an area of scratch space and then install a long trampoline to branch to the relocated code. We identify three sources of scratch spaces that can be used:

- Padding bytes. Compilers emit padding bytes to improve cache alignment. On `x86-64`, it has been shown that padding bytes are in general long enough for holding the 5 byte branch. However, on `ppc64le`and `aarch64`, the padding is at most three instructions long.
- Unused space in scratch blocks.
- Original sections that hold information needed for dynamic linking such as `.dynsym`, `.dynstr`, `.rela.dyn`.

With our trampoline placement analysis and trampoline instruction designs, the chance of needing trap based trampolines has been dramatically reduced. Still, we would use trap instructions as trampolines if necessary.

8 EVALUATION

We implement our new approaches as an extension to Dyninst and we will work with Dyninst developers to upstream our work. Dyninst provides programming APIs that allow users to choose arbitrary locations in a binary to instrument (called instrumentation

points) and allow users to inject arbitrary code to an instrumentation point.

We write a Dyninst program to verify the correctness and measure the overhead caused by incremental CFG patching. It instruments every basic block with empty instrumentation, which will trigger relocating all functions. It overwrites every code bytes in `.text` with illegal instructions and then installs trampolines at the addresses where our analysis determines necessary. This serves as a strong test to detect any mistakes in our binary rewriting.

8.1 SPEC CPU 2017

We run our test program with SPEC CPU 2017 to generate rewritten binaries and run the rewritten binaries 10 iterations with 8 threads on the following three systems: (1) a `x86-64`machine (Intel(R) Xeon(R) CPU E5-2695 v4 @ 2.10GHz), which has 72 processors and 128GB memory, and runs Red Hat 7.8 with gcc 7.3.0; (2) a `ppc64le`machine (POWER9), which has 160 processors and 256 GB memory, and runs Red Hat 7.7 with gcc 6.4.0; (3) a `aarch64`machine, which has 32 processors and 64 GB memory, and runs Ubuntu 19.04 with gcc 6.4.0.

The base and peak executables are compiled with -O3, `-no-pie` to generate position dependent code, and with the same compilation flags. Among the 20 total benchmarks, `627.cam4_s` does not compile on any of the systems, so is excluded. In the remaining 19 benchmarks, there are 8 benchmarks that are written in Fortran or contain Fortran components; other benchmarks are written in C/C++.

We only instrument the main executables. We evaluate three binary rewriting modes `dir`, which does not modify indirect control flow, `jt`, which modifies jump tables and direct control flow, and `func-ptr`, which modifies both direct and indirect control flow. All three modes have trampoline placement analysis and runtime RA translation enabled. We do our best to compare our approach with several existing tools, including Dyninst-10.2 (implementing SRBI) and Egalito (IR lowering) [1]. Our `dir`mode is essentially Dyninst-10.2 with our trampoline placement analysis and runtime RA translation. Egalito supports both `x86-64`and `aarch64`, but even with the help from a Egalito developer, we cannot get Egalito compiled on our `aarch64`system. For Egalito, we compiled the benchmarks with `-pie` as it does not support position dependent code.

Table 3 shows the results. First, our approaches can successfully instrument all SPEC CPU 2017 benchmarks. Dyninst-10.2 has 4 to 6 failed benchmarks. Two of the failed benchmarks use C++ exceptions. We find that while Dyninst-10.2 attempts to use call emulation to support C++ exceptions, this is only implemented on `x86-64`, not on `ppc64le`or `aarch64`. In addition, the call emulation on `x86-64`does not correctly handle indirect calls through stack memory locations. Egalito failed with the two benchmarks that use C++ exceptions, which is a known limitation.

Second, as we described in Section 5, if a function contains unresolved jump tables, the function will be marked uninstrumentable. Our results show that our approach significantly improves instrumentation coverage compared to Dyninst-10.2. Notably, on `x86-64`, Dyninst-10.2 marked over 10% of the total function uninstrumentable in the worse case, while our approach achieved 100% coverage on `x86-64`.

Third, we bridge the runtime overhead gap between code patching and IR lowering. While Dyninst-10.2 does not incur too high runtime overhead on average, we find that the failed benchmarks actually would incur prohibitive high runtime overhead when we fixed the bugs in Dyninst-10.2. We observed about 30% runtime overhead for the two benchmarks that use C++ exceptions after fixing call emulation and over 100% runtime overhead for `602.sgcc` after fixing signal delivery from Dyninst's runtime library to the application. Our three binary rewriting modes incrementally enable new techniques described in the paper and the results show that all our new techniques are effective in reducing runtime overhead. In particular, our `func-ptr`cuts the runtime overhead to close to zero. This shows that the performance advantages of IR lowering is rooted in being able to rewrite all control flow and code patching can also enjoy this close-to-zero runtime overhead if almost all control flow is rewritten. We observed a slight speedup from binaries rewritten by Egalito on average because Egalito enables some binary optimizations including reducing paddings between functions to improve icache efficiency. However, we also observed a 6.28% worse case overhead. We believe that the binary optimizations performed by Egalito are not necessarily always beneficial. For reference, E9Patch [15] reported 359.59% maximal and 110.81% on average runtime overhead using SPEC CPU 2006 on `x86-64`.

Next, we compare sizes of rewritten binaries. The results of size increase are calculated using the `size` utility from `binutils`. `size` counts only the sections that will be loaded into the memory at run time; so, unloaded sections such as debugging information are not counted. We show that our approaches generate similar sizes of rewritten binaries compared to other code patching based binary rewriting approaches. Overall, our approaches cause about 105% maximal and about 68% mean size increase on three architectures. Compared to Dyninst-10.2, our approaches generate slightly larger binaries on `x86-64`and `aarch64`because we store the return address mapping into the rewritten binaries, but significantly smaller binaries on `ppc64le`due to drastically smaller trap trampoline mapping. On `ppc64le`, Dyninst-10.2 tends generate many more trap trampolines compared to other architectures as the branch instruction on `ppc64le`has only 32MB branching range. For reference, E9Patch [15] reported 103.75% maximal and 57% on average size increase using SPEC CPU 2006 on `x86-64`.

We acknowledge that our approaches generates much larger binaries compared to Egalito. Essentially, we sacrifice binary size for better binary rewriting generality. The increased binary file can be a problem for embedded systems, where memory is a scarce resource, but should be a reasonable tradeoff for desktops, servers, and supercomputers.

Finally, we show that increased binary sizes do not lead to higher instruction cache misses in our approaches. If we suffer from high instruction cache misses, the runtime overhead for our approaches would not have been close to zero. In fact, a key design goal of `jt` and `func-ptr` modes is to reduce the bounce between original code and the instrumentation code, which will also reduce pollution to instruction cache caused by original code. In other words, while

[1] We also tried E9Patch, but it failed to generate valid binaries due to an engineering bug.

Table 3: Block level empty instrumentation results

	Time overhead		Coverage		Size increase		Pass
x86-64	max	mean	min	mean	max	mean	
SRBI	17.27%	5.48%	89.47%	97.37%	95.56%	62.56%	13
dir	16.77%	2.95%	100.00%	100.00%	101.53%	64.27%	19
jt	3.31%	1.35%	100.00%	100.00%	101.53%	64.27%	19
func-ptr	1.20%	0.18%	100.00%	100.00%	101.53%	64.27%	19
Egalito	6.28%	-0.68%	100.00%	100.00%	19.34%	-4.35%	17
ppc64le							
SRBI	12.73%	3.41%	93.55%	98.84%	191.74%	103.33%	15
dir	9.03%	1.27%	96.17%	99.41%	105.13%	69.17%	19
jt	8.91%	0.96%	96.17%	99.41%	105.13%	69.19%	19
func-ptr	3.86%	0.05%	96.17%	99.41%	105.13%	69.19%	19
aarch64							
SRBI	8.02%	0.81%	90.00%	97.34%	96.94%	64.11%	14
dir	12.38%	1.00%	99.91%	99.99%	109.30%	68.63%	19
jt	5.10%	0.36%	99.91%	99.99%	109.30%	68.63%	19
func-ptr	0.76%	-0.75%	99.91%	99.99%	109.30%	68.63%	19

our approaches increases code sizes, they do not increase the size of "hot code".

8.2 Real World Applications

Firefox's `libxul.so`: Firefox is written in multiple source programming languages. Its Rust code is all linked into `libxul.so`, along with its majority of C/C++ code. For Firefox 80.0, `libxul.so` has a 120 MiB `.text` section, which contains about 241K functions. We ran our test program to rewrite `libxul.so` and run Firefox with the rewritten `libxul.so` using two web browser benchmarks.

The first one is the Web Latency Benchmark [2], which measures the browser's responsiveness. We ran the benchmark 120 times on the `x86-64`machine. `jt`mode achieved 3.07% average overhead and 7.73% maximal overhead; `func-ptr`mode achieved 2.31% average overhead and 6.29% maximal overhead. `dir`mode failed because of a bug in the runtime library related to handling trap trampolines installed in library destructors. `jt`and `func-ptr`modes can reduce trap trampolines and thus avoids this problem and achieved 99.93% instrumentation coverage. The rewritten `libxul.so` binaries are 82.83% larger than the original one. Egalito ran into a segfault when rewriting `libxul.so` and an Egalito developer confirmed that it currently does not support some of the meta-data in Rust.

The second one is JetStream2 benchmark [1], which includes a variety of JavaScript and Web Assembly benchmarks and generates a single score that represents the performance of the web browser. We ran the benchmark 40 times on the `x86-64`machine. `jt`caused 2.08% average and 6.26% maximal score reduction. `func-ptr`caused 0.20% average and 5.92% maximal score reduction.

Docker: Docker is written in Go. We use our test program to instrument the docker executable (Docker version 19.03.11) under `/usr/bin` on the `x86-64`machine. We achieved 100% instrumentation coverage. In fact, Go's compiler does not emit jump tables for switch statement. Therefore, `dir`and `jt`are the same for Go binaries. `func-ptr`mode failed because of the language specific function tables in Go. Egalito cannot rewrite Go binaries due to unsupported meta-data and Go's builtin stack unwinding. We verified the correctness of our rewriting with 13 Docker's commands including `pull`, `run` and `exec`. We also ran several compound commands such as "`docker rm -f $(docker ps -aq)`" for deleting all containers. The rewritten binary incurred 6.98% average and 16.27% maximal overhead for these docker commands; it is 69.28% larger than the original binary. The maximal overhead of rewriting docker is significantly higher compared to `libxul.so` or SPEC CPU 2017; this shows the importance of being able to rewriting function pointers to reduce runtime overhead. We believe future function pointer analysis for Go binaries will help reduce this overhead.

These two large real world applications show that our approach can rewrite binaries written in modern programming languages, such as Rust and Go. Even though we cannot fully rewrite function pointers to achieve optimal runtime overhead, our approach is able to rewrite these binaries with small, additional overhead.

8.3 Comparison with BOLT

We performed a different experiment with SPEC CPU 2017 to compare our approach with BOLT [21]. We use Dyninst (with our approaches implemented) and BOLT to do two different code reordering on `x86-64`: (1) reverse all functions while keeping the order of blocks unchanged within a function, and (2) reverse all blocks in a function while maintaining the order of functions.

For (1), BOLT failed to reorder functions, printing error messages `BOLT-ERROR: function reordering only works when relocations are enabled`. We stress that BOLT emitted such message even for PIE and shared libraries, where there exist run time relocation entries. This is expected as BOLT requires link time relocation to be present to reorder functions. In contrast, our work can reverse functions for all SPEC CPU 2017 benchmarks.

For (2), BOLT successfully reordered blocks for 9 out of the 19 benchmarks but generated corrupted binaries for the other 10 benchmarks. These corrupted binaries have bad `.interp` data, causing them not be able to be loaded. The rewritten binaries emitted

by BOLT have 11% mean and 33% maximal size overhead. Our work can reorder blocks for all 19 benchmarks.

9 CASE STUDY

We use Diogenes [28, 29] as a case study, which employs binary instrumentation to identify unnecessary CPU/GPU synchronization and duplicated memory transfers. Prior research has shown that vendor supplied profiling interface may leave internal GPU activities unreported [28] and mislead users regarding where the true performance bottleneck is. Diogenes therefore does not rely on vendor interface, such as CUPTI from Nvidia, and it directly instruments user space GPU driver (`libcuda.so`) to collect performance information.

Diogenes developers identified that there is a stripped function inside `libcuda.so`, which is responsible for performing GPU synchronization. This internal function is called by all public interfaces that may trigger synchronization, such as `cuMemcpy`. CUPTI only reports synchronization activities that are initiated through public interfaces, while missing synchronization activities initiated internally. Therefore, to have a complete understanding of CPU/GPU synchronization, it is crucial to identify the internal synchronization function and instrument it to collect synchronization information.

Diogenes runs a test program to identify this internal synchronization function. This test program launches a GPU kernel that contains an infinite loop and performs synchronization to wait for the kernel to complete. As the kernel will never return, the internal synchronization function will also never return. In addition, Diogenes builds call graphs for known synchronization functions in the CUDA driver, such as `cuCtxSynchronize` and `cuMemcpy`. The internal synchronization function should be at the intersection of the call graphs. Diogenes instruments these common functions inside `libcuda.so` to trace whether a function returns or not. This instrumentation test generates typically 2 to 3 functions that do not return when synchronization is performed and Diogenes chooses the deepest function in the call stack as the internal synchronization function.

On `x86-64`, this instrumentation test takes 30 minutes to finish with Dyninst mainstream. With our improvement, the test takes 30 seconds. This 60X improvement is because our approach significantly reduces the use of trap-based trampolines.

We also tried to use Egalito to rewrite `libcuda.so`, but Egalito failed to generate a valid library due to not being able to rewrite symbol versioning information, which is commonly used by C++ libraries.

This example illustrates the value of partial instrumentation: Diogenes only needs to instrument 700 of the 12644 total functions in `libcuda.so`. Our approach can instrument these functions without being impacted by potential difficulties in other functions or difficulties related to meta-data.

10 DISCUSSION

Binary Analysis: The arm race between binary analysis and compiler code optimization is a lasting battle. We expect this battle to continue especially when Link Time Optimization (LTO) is becoming more popular. In general, our work and other binary rewriting approaches can readily benefit from advancements in jump table analysis, function pointer identification, and indirect tail call identification. In contrast to existing binary rewriting approaches where the analysis of failure modes is often missing, we have presented comprehensive analysis on how failures of binary analysis impact binary rewriting and new strategies to mitigate failures of binary analysis to improve binary rewriting

Dynamic Binary Instrumentation: Our approach can be extended to support dynamic binary instrumentation in a straightforward way. All the techniques described in this paper are applicable to dynamic instrumentation. One additional engineering effort to support dynamic instrumentation is the implementation of RA translation for C++ exception. During dynamic instrumentation, function wrapping through `LD_PREDLOAD` will not work. Function wrapping in the context of dynamic instrumentation can be done by rewriting the `.got` table.

Real-world Binary Rewriting Applications: In this work, we only evaluated our approaches with empty instrumentation; this is not what users of binary rewriting tools would do. We originally planned to evaluate runtime overhead of real tools such as function or block execution counts. We quickly found that the overhead of real tools is decided by two factors: (1) the binary rewriting infrastructure, and (2) how a user uses a binary rewriting infrastructure. We focused on (1) in this work. We believe (2) is an important but separate topic. As an example, the Dyninst project and Egalito both provides a sample tool that collects function execution counts. Egalito's sample tool runs much faster than the one from Dyninst. However, we found that the root cause of the overhead in the Dyninst's tool was caused by excessive function calls to an instrumentation library where execution counts are incremented, while Egalito's sample tool inserts inlined assembly to increment execution counts. In other words, one can use Dyninst to collect function execution counts in the same way as Egalito's sample tool and enjoys low overhead. We leave the topic of how to better use a binary rewriting tool as future work.

11 CONCLUSION

In this paper, we bridge the gap between existing IR lowering approach, which requires complete binary analysis, and instruction patching approach, which does not use any binary analysis. Our incremental CFG patching successfully balanced runtime overhead, generality, and instrumentation semantics. The foundation is a trampoline placement analysis. We defined control flow landing (CFL) blocks as the basic block where control flow can be transferred from instrumentation to the original code, and established that it is sufficient to install trampolines at only CFL blocks. We further reduced CFL blocks by rewriting jump tables and function pointers and designed runtime return address translation to support C++ exceptions and Go's runtime stack unwinding. We evaluated our new approach with SPEC CPU 2017 and `libxul.so` from Firefox, achieving small overhead. We also speeded up an instrumentation step in Diogenes, which instruments `libcuda.so`, from 30 minutes to 30 seconds.

We make a first step towards exploring how failures from binary analysis would impact binary rewriting and believe that this is a good methodology for reasoning about reliability of binary rewriting. Our approach currently does not guarantee instrumentation

safety as we use new heuristics to improve indirect tail call identification, aiming for higher instrumentation coverage. We believe that principled indirect tail call identification is an interesting future research direction to augment our binary rewriting approach.

ACKNOWLEDGMENTS

We would like to thank our shepherd Ding Yuan and anonymous reviewers for their insightful comments.

A ARTIFACT APPENDIX

A.1 Abstract

Our artifacts consist of an extension to Dyninst that implements the new binary rewriting approach described in the paper, test programs that use Dyninst to instrument SPEC CPU 2017, Firefox's `libxul.so`, and the Docker executable, and related scripts for running the experiments.

A.2 Artifact Check-list (Meta-information)

A SPEC CPU 2017 benchmark suite (version 1.0.2) is needed for this Artifact Evaluation.

- **Compilation:** Any version of GCC (we use the system compiler to build required version of GCC through spack)
- **Transformations:** Binary rewriting using Dyninst.
- **Run-time environment:** Root access to Ubuntu Linux with both terminals and GUI. We recommend Ubuntu Bionic 18.04.
- **Hardware:** A `x86-64` machine (Intel Xeon E5-2695 v4 recommended), which has 72 processors and 128GB memory; a `ppc64le` machine (POWER9), which has 160 processors and 256 GB memory; a `aarch64` machine, which has 32 processors and 64 GB memory.
- **Metrics:** We use execution time to measure instrumentation overhead and percentage of instrumented functions as instrumentation coverage.
- **Output:** For web browser based benchmarks, results are shown on the browser. Otherwise, the results are printed in the console.
- **Experiments**: Using Bash scripts and Linux commands provided.
- **How much disk space required (approximately)?:** 50GB.
- **How much time is needed to prepare workflow (approximately)?:** Two hours.
- **How much time is needed to complete experiments (approximately)?:** A couple of days, which can be reduced by running few iterations of SPEC CPU 2017.
- **Archived (provide DOI)?:** 10.5281/zenodo.4540633

A.3 Description

A.3.1 How to Access. The artifacts are available on GitHub: https://github.com/mxz297/Incremental-CFG-Patching-ASPLOS21-AE.

The repository provides a README file that describes the detailed steps to evaluate the artifact. We briefly summarize the steps below.

A.3.2 Hardware Dependencies. The SPEC CPU 2017 experiment can be run on any one of the `x86-64`, `ppc64le`, and `aarch64` architectures. The web browser and Docker experiments should be run on `x86-64`.

A.3.3 Benchmarks. The benchmarks of our experiments are executables as follows.

(1) SPEC CPU 2017 (version 1.0.2)
(2) Firefox (version 80.0) with the Web Latency Benchmark and the Jetstream2 Benchmark
(3) Docker (version 19.03.6)

A.4 Installation

You can get the artifacts and its dependencies according to the detailed README file from our AE Github repository mentioned above.

Please follow the "Setup software" section available at https://github.com/mxz297/Incremental-CFG-Patching-ASPLOS21-AE#setup-software and run the setup bash script `build-x86.sh` when running on `x86-64`.

A.5 Experiment Workflow

The overall workflow consists of the following steps:

(1) Install the artifact and dependencies
(2) Run the SPEC CPU 2017 experiment
(3) Run the Firefox experiment
(4) Run the Docker experiment

We provide scripts for each of the steps above.

A.6 Evaluation and Expected Result

A.6.1 SPEC CPU 2017.

Installation: In directory `setup`, there are building scripts such as `build-x86.sh`. Run the corresponding script based on the architecture. Besides installing necessary software dependencies, it will generate a file `spec-config-paths.txt`, which contains the paths needed to copy and paste into the configuration file for running SPEC CPU 2017.

Evaluation: In directory `spec2017`, we provide several scripts and a template SPEC configuration file.

Please refer to https://github.com/mxz297/Incremental-CFG-Patching-ASPLOS21-AE#spec-2017 for detailed instructions to prepare the SPEC configuration file. We provide a script `run_spec.sh` for each architecture to run SPEC experiments (For `x86-64`, it is `x86/run_spec.sh`). and a script `run_result.sh` to print the results to the console. Detailed instructions and example runs for these scripts are available in the README file.

A.6.2 Firefox.

Installation: Firefox (version 80.0) is usually shipped with the latest Ubuntu 18.04 dist. To install it manually, one can visit https://support.mozilla.org/en-US/kb/install-firefox-linux and choose the version 80.0 for this evaluation.

In directory `firefox`, we provide scripts to instrument Firefox's `libxul.so` and prepare environments to run the instrumented version. `run_inst.sh` will instrument `libxul.so` and `source env.sh`

to prepare the environments. Then, replace the original `libxul.so` with the instrumented one.

Please refer to https://github.com/mxz297/Incremental-CFG-Patching-ASPLOS21-AE#firefoxs-libxulso for detailed instructions.

Evaluation: We provide two web browser based benchmarks for evaluation.

• *Web Latency Benchmark.* Download the benchmark from http://google.github.io/latency-benchmark/latency-benchmark-linux.zip. Follow the README file inside the zip file to run Web Latency Benchmark, and collect the results displayed in the web browser.

• *Jetstream2 Benchmark.* Type https://browserbench.org/JetStream/ in Firefox search box. Click the 'Start Test' button to run Jetstream2, and collect the results displayed in the web browser.

A.6.3 Docker Experiment.

Installation: Docker Installation guide can be found at https://docs.docker.com/engine/install/ubuntu/.

Evaluation: In directory `docker`, we provide scripts to instrument the docker executable and prepare environments to run the instrumented version.

`run_inst.sh` will instrument the docker executable and `source env.sh` to prepare the environments. Then, replace the original `docker` with the instrumented one. Please refer to https://github.com/mxz297/Incremental-CFG-Patching-ASPLOS21-AE#docker-executable for detailed instructions. Then, run `run_docker.sh` to test the rewritten docker executable. here should be no errors.

A.7 Notes

Please exercise with cautions when replacing original binaries (e.g., libxul.so and docker). Always prepare a backup for the evaluation.

REFERENCES

[1] [n.d.]. JetStream 2 - BrowserBench, https://browserbench.org/JetStream/.
[2] [n.d.]. Web Latency Benchmark, https://google.github.io/latency-benchmark/.
[3] Dennis Andriesse, Xi Chen, Victor Van Der Veen, Asia Slowinska, and Herbert Bos. 2016. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*. 583–600.
[4] Théophile Bastian, Stephen Kell, and Francesco Zappa Nardelli. 2019. Reliable and Fast DWARF-Based Stack Unwinding. *Proc. ACM Program. Lang.* 3, Article 146 (Oct. 2019), 24 pages.
[5] Erick Bauman, Zhiqiang Lin, and Kevin W Hamlen. 2018. Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics. In *Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, USA.
[6] M. Ammar Ben Khadra, Dominik Stoffel, and Wolfgang Kunz. 2020. Efficient Binary-Level Coverage Analysis. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)* (Virtual Event, USA). 1153–1164. https://doi.org/10.1145/3368089.3409694
[7] Andrew R. Bernat and Barton P. Miller. 2012. Structured Binary Editing with a CFG Transformation Algebra. In *2012 19th Working Conference on Reverse Engineering (WCRE)*. Kingston, ON, Canada, 9–18.
[8] Andrew R. Bernat, Kevin A. Roundy, and Barton P. Miller. 2011. Efficient, Sensitivity Resistant Binary Instrumentation. In *The International Symposium on Software Testing and Analysis (ISSTA)*. Toronto, Canada.
[9] BOLT. [n.d.]. Fix C++ exceptions for shared objects, https://github.com/facebookincubator/BOLT/commit/57e6864676195b7d883ebde59437e3de19d6181b.
[10] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. 2003. An Infrastructure for Adaptive Dynamic Optimization. In *International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO)*. San Francisco, California, USA.
[11] Buddhika Chamith, Bo Joel Svensson, Luke Dalessandro, and Ryan R Newton. 2017. Instruction punning: Lightweight instrumentation for x86-64. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 320–332.
[12] Thurston HY Dang, Petros Maniatis, and David Wagner. 2015. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. ACM, 555–566.
[13] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. 2017. Rev.Ng: A Unified Binary Analysis Framework to Recover CFGs and Function Boundaries. In *26th International Conference on Compiler Construction (CC)*. Austin, TX, USA.
[14] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. 2020. RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization. In *41st IEEE Symposium on Security and Privacy (Oakland)*.
[15] Gregory J. Duck, Xiang Gao, and Abhik Roychoudhury. 2020. Binary Rewriting without Control Flow Recovery. In *41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. London, UK.
[16] Yizi Gu and John Mellor-Crummey. 2018. Dynamic Data Race Detection for OpenMP Programs. In *International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*. Dallas, Texas.
[17] Yuanbo Li, Shuo Ding, Qirun Zhang, and Davide Italiano. 2020. Debug Information Validation for Optimized Code. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. London, UK, 1052–1065. https://doi.org/10.1145/3385412.3386020
[18] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. Chicago, IL, USA, 190–200.
[19] Xiaozhu Meng, Jonathon M. Anderson, John Mellor-Crummey, Mark W. Krentel, Barton P. Miller, and Srđan Milaković. 2020. Parallel Binary Code Analysis. arXiv:2001.10621 [cs.PF]
[20] Xiaozhu Meng and Barton P. Miller. 2016. Binary Code Is Not Easy. In *The International Symposium on Software Testing and Analysis (ISSTA)*. Saarbrücken, Germany.
[21] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. 2019. BOLT: A Practical Binary Optimizer for Data Centers and Beyond. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. Washington, DC, USA, 2–14.
[22] Paradyn Project. [n.d.]. Dyninst: Putting the Performance in High Performance Computing, http://www.dyninst.org.
[23] Red Hat. accessed Aug. 12, 2020. Product Life Cycles. https://access.redhat.com/product-life-cycles.
[24] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. 2016. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*. San Jose, CA, USA.
[25] V. v. d. Veen, E. Göktas, M. Contag, A. Pawoloski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida. 2016. 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.
[26] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. 2015. Practical Context-Sensitive CFI. In *22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*. Denver, Colorado, USA.
[27] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. 2017. Ramblr: Making reassembly great again. In *24th Annual Symposium on Network and Distributed System Security (NDSS)*. San Diego, CA, USA.
[28] Benjamin Welton and Barton P. Miller. 2019. Diogenes: Looking for an Honest CPU/GPU Performance Measurement Tool. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC'19)*.
[29] Benjamin Welton and Barton P. Miller. 2020. Identifying and (Automatically) Remedying Performance Problems in CPU/GPU Applications. In *34th ACM International Conference on Supercomputing (ICS)*. Barcelona, Spain, Article 27, 13 pages.
[30] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P. Kemerlis. 2020. Egalito: Layout-Agnostic Binary Recompilation. In *Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Lausanne, Switzerland.
[31] David Williams-King and Junfeng Yang. 2019. CodeMason: Binary-Level Profile-Guided Optimization. In *3rd ACM Workshop on Forming an Ecosystem Around Software Transformation* (London, United Kingdom) *(FEAST'19)*.
[32] Mingwei Zhang and R. Sekar. 2013. Control Flow Integrity for COTS Binaries. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)* (Washington, D.C.).