Binary Rewriting without Relocation Information

Technical report, University of Maryland, November 2010

Matthew Smithson Kapil Anand Aparna Kotha

Khaled Elwazeer Nathan Giles Rajeev Barua

Abstract

Binary rewriting softwares transform executables by maintaining the original binary's functionality, while improving it in one or more metrics, such as runtime performance, energy use, memory use, security, and reliability.

Existing static binary rewriters are unable to rewrite binaries that do not contain relocation information, which is typically discarded by linkers unless specifically instructed otherwise. Unfortunately, most deployed binaries lack such information; hence they cannot be statically rewritten at all.

We present a new approach to binary rewriting capable of rewriting binaries statically *without* relocation information. This is the first technology that allows for application of complex static transformations to any binary. This technology extends the power of binary rewriting technology past developers and into the hands of end-users.

This technology has been incorporated into a working prototype known as *SecondWrite*, which has been used to rewrite and apply optimizations to a subset of SPEC benchmarks. Tests yielded reasonable overheads, with an average speedup of 27% for non-optimized binaries, and an average slowdown of 7% for optimized versions. Our tool can be used as a platform for complex optimization and security enhancements of any binary for the first time ever.

Categories and Subject Descriptors D.3.4 [*Software*]: Processors—Compilers, Translator writing systems and compiler generators

General Terms Performance, Theory

Keywords binary rewriting, relocation information, optimization, instrumentation, disassembly, reverse engineering

1. Introduction

Binary rewriting is a promising technology that is finding widespread application in new research ideas being published. Researchers have proposed binary rewriting-based methods for a variety of topics, including inter-procedural optimization [12, 24, 28], security-policy enforcement [14], preventing control-flow attacks [5, 27], cache optimization [23], software caching [19], and distributed virtual machines for networked computers [31].

The reason for the great interest in research in binary rewriting is that it offers many additional advantages over a traditional compiler toolchain:

Whole-program Optimizations. Build systems for statically-compiled programs typically rely upon separate compilation in order to minimize compile times, limiting the efficacy of any available compiler-based whole-program optimizations. In contrast, binary rewriters operate on programs after the compilation units have been merged together, leaving them more naturally suited for performing whole-program optimizations.

Transformation Coverage. Unlike compilers, binary rewriters can transform statically linked library code, as well as code originally written in assembly. [1]

Transformation Reuse. Binary rewriters do not differentiate between source language or compiler. It is thus more efficient to implement a code transformation once for a binary rewriter, rather than repeatedly within various compilers.

Security Enforcement. A binary rewriter can be used by an end-user to insert security checks into a binary compiled by an untrusted source.

Source Code Requirement. Binary rewriting technology can operate in environments where source code is unavailable, such as legacy, and third-party binaries.

[1] Some compilers, including GCC, work around this limitation in some cases by providing 'built-in' versions for certain library functions.

Additional Optimization. Binary rewriters can further improve previously optimized binaries by applying optimizations that were either missed or were unavailable during the original compilation.

Platform-Aware Optimizations. Binaries compiled for distribution are often targetted for a particular ISA, but are rarely optimized for a particular processor. Binary rewriters can apply customized optimizations to take advantage of processor-specific characteristics such as the availability of additional instructions (such as multimedia extensions) and knowledge about the memory hierarchy and processor pipeline depths.

Consequent to the advantages of binary rewriting, a number of binary rewriters, disassemblers, and link-time optimizers have been designed. Existing tools typically fall into one of several categories: *static* rewriters, *dynamic* rewriters, and minimally-invasive rewriters. A static binary rewriter (like ours) rewrites the binary off-line, but requires relocation (and usually symbolic) information. A dynamic binary rewriter rewrites the binary during its execution, but, in a key advantage, does not need relocation or symbolic information. Let us discuss each in turn.

Static binary rewriters [7–9, 12, 15, 16, 24, 28, 32, 34], sometimes referred to as link-time optimizers, rewrite code offline. Given their offline nature, they have the time to perform complex analysis and transformations. However, they require relocation information in order to both identify code regions and address locations. The identification of code regions is required in order to ensure complete disassembly of a binary. It is invalid to merely disassemble the entire code segment, as compilers often embed data (such as jump tables, alignment bytes, literal tables, and padding bytes) throughout the code segment. The identification of address locations is required in order to adjust for movement of the address target during the rewriting process.

Minimally-invasive rewriters [2] do *not* require relocation information. These tools maintain the original machine code in place to the greatest extent possible. However, as a result, these rewriters typically support only minimal transformations such as the insertion of 'trampolines' to jump into and out of instrumentation blocks [1] and peephole optimizations affecting small sequences of instructions. Our goal is to provide a platform that is capable of complex whole-program optimizations that can revise the code in any manner they choose. Minimally invasive rewriters cannot do this.

Dynamic rewriters [1, 4, 6, 22, 25, 26, 30] *are* able to rewrite without relocation information. These tools perform disassembly, analysis, and transformation at runtime, at which point the identification of address locations and code regions is trivial.

Unfortunately, dynamic rewriters tend to suffer from significant runtime overhead, even when performing no transformations because of the overhead of rewriting at run-time. When performing complex transformations such as automatic parallelization and inter-procedural optimizations, the overhead is is likely to be prohibitive. As a result, dynamic rewriters have thus far only been used for simple instrumentation and peephole optimization. For example, some dynamic rewriters intercept the application's execution at most indirect control transfers in order to ensure that the target has been rewritten. As a result, even without applying any transformations, reported overheads range from 20% for DynamoRIO [6] to 54% for PIN [22]. Nevertheless, dynamic rewriters have seen some commercial success such as in the use of DynamoRIO by Determina Inc. for its security checks on control-flow, because of their applicability to arbitrary binaries without relocation information. However it is unlikely that dynamic rewriters will ever be able to perform complex transformations of the types we want without incurring prohibitive run-time overheads.

Consequent to the limitations of dynamic and minimally-invasive rewriters, the promise of a successful static rewriter is as great as ever. However, the reality of binary rewriting has been far more disappointing. *Despite the tremendous potential, static binary rewriting is not in widespread commercial use today*. Binary rewriters are not included in most compilation toolchains as a final step. Most end-users are likewise not rewriting programs, since widely used commercial binary rewriters do not exist.

The reason for the lack of success of static rewriting technology is clear: all existing static binary rewriters require relocation information. [2] [3] Unfortunately this information is discarded by linkers by default. As a result, binaries do not contain this information unless it is explicitly included by the developer. We surveyed ten widely used commercial binaries: Adobe Acrobat Reader, AOL Instant Messenger, Corel Photo Album 6, Microsoft Word 2002, Microsoft Powerpoint 2002, Post-It Software Notes Lite 3.1.1, Putty 0.58, QuickTime Player 6.5, Spybot Search & Destroy 1.4, and TextPad 4.7.3. We found that *none* of these ten executables contained relocation or symbolic information. Hence, these executables cannot be rewritten by any existing static rewriter, making existing static rewriter use impractical.

Linkers typically do provide a way to retain relocation information by using special command-line flags. [4] However, since only the original developers have access to the object code, only they can request that this information be retained. *The end result is that only the original developers can rewrite their binaries.* Binary rewriting by end-users on arbitrary binaries is impossible today.

We present the first static binary rewriting technology ever that is capable of rewriting binaries without relocation

[2] Unless otherwise specified, we exclude minimally-invasive rewriters when referring to static binary rewriters

[3] Relocation information includes ELF relocation entries or other equivalent formats

[4] For example, GCC's ld linker provides the *–emit-relocs* flag to output relocation information.

information. This implies that, for the first time, an arbitrary user will be able to statically rewrite any arbitrary binary. Our technology supports unrestricted wholesale recompilation of the binary, including instruction (re)selection, register (re)allocation, and standard optimizations. Unlike dynamic rewriters, our technology introduces reasonable runtime overheads. Unlike minimally-invasive rewriters, our technology supports arbitrarily complex transformations such as automatic parallelization. This technology provides unprecedented power to end-users to rewrite their programs to improve performance, energy use, memory consumption, security, and reliability.

Our static binary rewriting technology has the following advantages over existing static binary rewriters:

- **Does not require relocation information**. Existing binary rewriters require developers to retain relocation information inside the input binary. Our rewriter eliminates this restriction, allowing anyone to rewrite any binary executable.
- **Can be applied to legacy applications**. Legacy binaries cannot be rewritten by existing binary rewriters since virtually all binaries lack relocation information. Moreover recompilation from source is often not possible since source code is often not readily available for legacy code.
- **Can be used to enforce security on untrusted code**. Since existing static binary rewriters can only be used with developer cooperation, it is not feasible to use them to enforce security properties on code from untrusted developers.

Our technology has been implemented in *SecondWrite*. SecondWrite is a static binary rewriter which targets the x86 ISA and integrates into the LLVM [21] compiler infrastructure. Test results from rewritten SPEC 2006 benchmarks [5] confirm our technology to be a platform for static rewriting of arbitrary binaries with reasonable runtime overhead.

2. Binary Rewriting Without Relocation Information

At first glance it seems impossible to statically rewrite a binary without relocation information. After all, if even a single branch in the program has an unknown target, it could branch to anywhere; hence no instruction could be moved from its current location, preventing rewriting altogether. This thinking has gone unchallenged for two decades, preventing the promise of binary rewriting from transforming itself to a practical, widely-used commercial reality. We propose methods that will allow arbitrary binaries to be statically rewritten for the first time.

All binary rewriting technologies must overcome two main challenges: relocating instructions and code discovery. This section examines each challenge, explains how traditional static rewriting technology overcomes those challenges, and presents solutions which do not require supplemental information.

[5] With the exception of equake, which was taken from the SPEC OMP 2001 benchmark suite.

2.1 Relocating Instructions

Certain instructions in a binary will contain operands which reference other locations also within the binary. Load and store instructions reference locations containing data. Branches and calls reference locations containing other instructions. During the rewriting process, the targets of these references move around. A binary rewriter must account for this movement. Rewriters must first identify the locations of these references, which we refer to as *address creation points (ACPs)*, and secondly adjust the addresses to account for any movement.

In some cases, the identification of ACP locations is trivial. Consider direct control transfer instructions (CTIs), such as direct branches and direct calls. For these instructions, the ACP is found directly following the instruction opcode in the binary. However, for indirect CTIs, such as an indirect call, the ACP is decoupled from the opcode. The ACP may exist in some remote location in the binary, where it is passed through a variety of mechanisms (via registers, memory, function arguments, global variables, etc.) to the usage site.

Thus, the identification of ACPs for indirect CTI is nontrivial. To overcome this problem, assisted static rewriters rely upon supplemental information. Often, this supplemental information comes in the form of *static relocation entries*. Because absolute addresses are not computable until the linking phase, the compiler will instead generate a relocation entry wherever an absolute address is required. These entries direct the linker to generate absolute addresses at certain locations within the binary. As a result, the set of relocation entries reveals all of the absolute ACPs within the binary. For those ISAs, such as Intel's x86, which do not support PC-relative indirect addressing for CTI instructions, this list of absolute ACPs is sufficiently comprehensive. Thus, to account for instruction (and data) relocation, assisted static rewriters can simply iterate across the list of relocation entries, updating each ACP with the associated address in the rewritten binary.

Our goal is binary rewriting without relocation entries. Let us consider how that might be done. Without relocation entries, an initial approach would be to scan the binary for instruction operands that appear to be addresses. Unfortunately, because a binary contains no information about operand types, it is difficult to determine whether an operand represents a constant data value or the address of an object. Consider the following move instruction:

```
8200: mov $0x8900, %eax
```

Assume that in the original binary, address `0x8900` corresponds to the base address of functino `foo`, and that function `foo` was subsequently rewritten to a different location. It would be unsafe to modify the above move instructions' operand to point to the new location of `foo`, unless we can somehow prove that the operand actually represents an address. If we did choose to update operand, but the operand instead was representative of a data value (perhaps a loop bound), then the correctness of the rewritten program would be invalidated. On the otherhand, if we choose *not* to update the operand, but it actually *did* represent the address of `foo` (an indirect call operand), then program's correctness will also be invalidated, as the rewritten operand would point to `foo`'s original (now incorrect) location. Thus, in order to take the same approach as assisted rewriters and update ACPs statically, we must be able to definitively prove a value to be an address and not a constant data value.

We handle indirect address translation by updating the address *usage* point instead of the address creation point. Unlike static ACP translation, this approach avoids the requirement of definitively identifiying ACPs in the binary altogether. Identification of indirect address usage points is trivial, as these instructions are readily revealed by their opcodes. In order to adust the address operands for these instructions, we introduce the notion of a *translator*.

Translators are comprised of code that is inserted directly into the intermediate representation just prior to every indirect CTI. Translators examine the indirect CTI operand and provide an appropriate adjustment to effectively translate the original address into the corresponding address in the rewritten binary.

Consider the following indirect call:

```
call *fp;
```

In the intermediate representation for the rewritten binary, the indirect call would be prefaced with its associated call translator. Consider the following example of a 2-entry call translator. Larger target sets are accommodated by simply adding additional cases.

```
if (fp == 0x8300):
    fp_modified = &fwrite;
if (fp == 0x8400):
    fp_modified = &fread;
call *fpmod;
```

To guarantee correctness, a translation must be provided for *every* possible target of the indirect CTI. Before we discuss how these target sets are generated, it is significant to first point out that the usage point translation approach allows for inclusion of extraneous translations *without sacrificing correctness*. Extraneous targets are those which the associated indirect CTI never actually targets. Assume in the previous example that fread is not an actual target for fp. In this case, including fread in the translator is useless, as that particular translation will never be executed. However, the presence of the fread translation does not jeopardize correctness. This notion is important, because it allows us to construct the target list for each CTI in a conservative manner.

We will leverage this feature by assuming, for now, that an indirect CTI may target *any* valid location. This implies that an indirect branch might target any block within its containing function, and that an indirect call might target any function in the program.

Clearly, usage point translation will introduce runtime overheads unlike the static ACP translation approach. Additionally, our the very conservative approach to indirect CTI target set identification may further increase translation times. Section 3 outlines techniques used to reduce these overheads.

2.2 Relocating Data

The previous section examined how instruction references can be adjusted to account for movement during rewriting has been examined. However, the presented solution did not address indirect *data* references. Our approach is to simply avoid the issue by prohibiting movement of any data targets during the rewriting process. This is realized by maintaining the original data segments in the rewritten binary for subsequent loading to their original address locations. The original code segment is also preserved in a similar fashion, as it may contain embedded data. Note that this leads to some duplication in that the original code segment will exist in the rewritten binary alongside its functionally-equivalent rewritten copy.

2.3 Code Discovery

In addition to instruction relocation, binary rewriters must overcome the challenge of identifying which portions of the binary contain instructions. This process, which we refer to as code discovery, is complicated by the fact that data is often embedded within the code segment. Data can appear in the code segment for a variety of reasons, including jump tables, padding bytes, alignment bytes, and literal tables. Thus, it cannot be assumed that the code segment is comprised solely of instructions alone.

One code discovery algorithm, known as *linear sweep* [10, 16], marches through a region, disassembling each location in a linear fashion. However, this algorithm will disassemble past unconditional branch instructions. This can lead to situations where the algorithm disassembles past an instruction and into a region of embedded data, (incorrectly) disassembling the contents of the data region as if it contained instructions.

A more appropriate code discovery algorithm for binary rewriting is *recursive traversal* [10], which discovers code by following only valid control flow edges. When a CTI is encountered, recursive traversal continues discovery at the CTI target locations, rather than at the subsequent file offset. Unfortunately, the targets of *indirect* CTI are not readily identifiable statically. As a result, recursive traversal cannot continue discovering code past these instructions.

Static rewriters are able to overcome this limitation of recursive traversal by identifying indirect CTI targets and restarting the discovery process at those locations. The previous section discussed relocation entries reveal address creation points in the binary. Note that the *contents* of these ACPs reveal the set of indirect CTI targets in the binary. Thus, assisted static rewriters can rely upon relocation entries (or their equivalent) to guarantee complete code discovery.

However, our goal is to perform complex transformations on arbitrary binaries, requiring complete code discovery without access to relocation entries. Previously, we assumed that indirect CTIs could target *any* location. We indicated that this conservative approach might produce extraneous translations, but would not sacrifice correctness. However, the implications for disassembly are more complex.

Extraneous CTI targets implies that disassembly will occur at locations not necessarily guaranteed to be targets, and which in some cases may actually not contain valid instructions at all. For these cases, we perform *speculative* recursive traversal disassembly, where a portion of the binary is disassembled as if contained instructions, even though that is not guaranteed to be the case.

Other research has used speculative techniques for code discovery in order to increase code coverage. However, our method is the first to incorporate speculation in a way that both guarantees 100% disassembly coverage *while also maintaining correctness*. Suppose a portion of data embedded within the code segment is mistakenly identified as an indirect CTI target. It will be speculatively disassembled as if it contained instructions. A translation would be inserted in the associated translator, pointing to the newly-disassembled instructions in the IR. Because the target region was actually data, the original indirect CTI could not have actually targetted the location. Thus, in the rewritten binary, the translator will never redirect execution to the speculatively-disassembled sequence. Additionally, as mentioned previously, we maintain a copy of the original code segment in place in order to guarantee in order to guarantee that any data references to this region will also maintain their correctness.

The following sections will discuss mechanisms for reducing the target set size for a given indirect CTI. However, it should be noted that the speculative disassembly process can also help in this regard through the identification of invalid speculative code sequences. Invalid sequences are identified as violating certain characteristics of well-formed code, such as containing control flow inconsistent with known code (non-speculative) sequences. For example, a speculative sequence containing a branch into the middle of a known code instruction would qualify as containing inconsistent control flow. Additionally, encountering an invalid opcode would be sufficient to classify a sequence as invalid. Once identified, invalid sequences are pruned from the intermediate representation and removed from their associated translators.

2.4 Callbacks

Previously, data references were addressed by avoiding the movement of data alltogether. In addition, indirect instruction references were handled by applying usage point translation. We have thus far assumed the address usage point to lie within the binary, allowing for the insertion of a usage point translator. However, situations do exist where the usage point lies outside the bounds of the input binary.

Function pointers may be passed as to libraries in order to register the function to be called at a later time. During the process of rewriting, unless additional information is known about the library, it may be impossible to determine whether the parameter is indeed a function pointer, or merely a constant value coincidentally equivalent to some function's virtual address.

One way to guarantee correctness in these situations is by leaving the parameter unmodified. In cases where the parameter was a coincidental constant value, this solution works because the original location will remain unchanged. In cases where the parameter was in fact a function pointer, control flow will return back to the original copy of the function, due to our restrictions on maintaining original copies of the code and data segments in their original locations (see section 2.2).

In order to guarantee correctness in the presence of callbacks, we allow control flow to return to the original copy of the code section. It is desirable to then re-transfer control back to the associated rewritten function, as soon as possible.

For those functions definitively classified as code during disassembly, we can replace the first instruction bytes with an unconditional jump to the location of the rewritten copy of the function. If the first few bytes of the function are not large enough to contain the jump (that is, a non-instruction byte is encountered), we can replace the first byte with a software interrupt instruction, as is done some in binary instrumentation tools [1]. The interrupt allows control flow to jump to an interrupt handler, which performs the control flow transfer to the rewritten copy of the function. *Thus, if control flow is ever directed to the original copy of a function, it will immediately return to the rewritten copy.*

There still remains the possibility of callbacks to speculative functions. Because these functions have not been definitively classified as code, it is unsafe to update their contents. In these situations, if the library function calls the parameter as a function pointer, control flow remains in the original binary until it is returned back to the library, or a non-speculative function is called.

2.5 Limitations

Thus far, we have presented an approach for performing static binary rewriting without relocation information. This approach guarantess correctness of any rewritten binary, aside from two limitations. First, like most static binary rewriters, self-modifying code is not handled. However, this is not a serious limitation since most modern operating systems, including Microsoft Windows [3], prohibit both self-modifying code and code execution from within the data segment for security reasons.

Additionally, binaries which have been modified to obfuscate control flow are not properly rewritten by this technology. Specifically, this technology relies upon the recursive traversal discovery algorithm, and assumes that CTIs exhibit normal behavior. For example, it is assumed that both targets of a conditional branch instruction will always contain valid instructions.

3. Optimizing Target Sets

Section 2 presented an approach for performing static binary rewriting without relocation information. Though the technology guaranteed correctness of the rewritten binary, it is extremely conservative in identifying indirect CTI targets.

Clearly, extraneous targets will lead to increases in code size. However, they are also disadvantageous to runtime performance by increasing the time required for usage point translation and limiting the effectiveness of optimizations by introducing unnecessary control flow edges. To address these concerns, this section introduces techniques for reducing the size of indirect CTI target sets through the elimination of false targets.

3.1 Constant Propagation

Indirect call target discovery via constant propagation is a technique used by DeSutter et. al. [10]. Some indirect control transfer targets can be traced to their usage sites by performing constant propagation (a type of dataflow analysis). It was discovered that the targets of 92% of indirect calls could be discovered via constant propagation. However, these particular results are heavily reliant upon the Alpha architecture, where all intermodular calls are made via indirect CTI. As a result, a vast number of indirect calls have only a single target. Unfortunately, compilers for other architectures, such as x86, tend to introduce indirect calls only when multiple targets are possible. In these situations, constant propagation alone is not sufficient to fully characterize the target set.

3.2 Binary Characterization

We have developed a new technique to effectively eliminate the vast majority of presumed indirect CTI targets. Our technique, termed *Binary Characterization*, leverages the restriction that indirect CTI require an absolute address operand, and that these address operands must appear within the code and/or data segments. As discussed previously, in a stripped binary without type information, it is not always possible to prove whether a data location is an address (and not constant data). However, it *is* sometimes possible to to prove that a location is *not* an address. Thus, it is possible to generate a list of values that *may* represent addresses. This address list will be guaranteed to be a superset of the actual list of indirect CTI targets.

Binary Characterization generates this list of possible addresses by first constructing a valid address range. The executable provides both the base virtual address and the size of the code segment. Together, these values form the basis for the binary's virtual address range. The code and data segments are subsequently scanned for values that lie within the constructed range, taking into account the endianness and native address size of the underlying instruction set architecture. The result is a list of values guaranteed to contain, at a minimum, all of the indirect CTI targets. Although the list may still contain extraneous targets, Binary Characterization can still eliminate a significant number of potential targets.

Additionally, this list can be further optimized by eliminating entries that are definitively known not to be data. Entries generated from known instruction opcode bytes, or known direct CTI operands can be eliminated, as we know for certain that these locations do not represent indirect CTI operands. Also, those entries which exhibit contradictory control flow (by pointing to the middle of known instructions) can be eliminated as well.

It should be noted that this technique is only appropriate for reducing target sets in those situations where addresses are not calculated at runtime. Thus, it is appropriate for indirect calls, but not always for indirect branches, where jump table implementations *can* contain runtime-computed addresses.

3.3 Alias Analysis

Constant propagation is sometimes able to propagate indirect CTI operands directly to their usage sites in rare cases where the indirect CTI has only a single target. However, most indirect CTI exist specifically *because* they contain multiple targets can cannot be effectively expressed as a direct CTI. In these situations, a more robust analysis is required for tracing operands to their usage sites.

Indirect CTI operands may be passed through the program in a variety of ways including memory, stack, function arguments, and registers. Alias analysis is capable of tracing operands through these mechanisms. We employ alias analysis by examining each indirect CTI operand against each entry in the reduced set of CTI targets provided by Binary Characterization. This allows for the elimination of CTI targets that are guaranteed not to alias a particular CTI operand. In cases where alias analysis discovers that an indirect branch operand must alias a set of targets, those entries not included in the 'must alias' set can be eliminated from the indirect branch's translator.

Further optimization is possible in some situations. When alias analysis discovers a single target that must alias a particular indirect CTI operand, the indirect CTI can be promoted to a direct CTI. Additionally, in situations where a set of targets is known to 'must alias' an operand, traditional static address point translation can be used in favor of usage point translation.

3.4 Indirect Branches

Binary Characterization is not appropriate for the discovery of indirect branch targets in situations where addresses may be calculated in the binary. As a result, additional techniques are necessary for reducing these target sets. One simple technique is to limit branch targets to within the current function boundary. This requires establishment of function boundaries, which can be complicated by the presence of functions containing multiple entry points.

Another technique, commonly employed by other rewriters, is to use pattern matching to identify the bounds of jump tables. Unfortunately, jump table bounds checks are compiler specific, so a single pattern does not suffice.

3.5 Function Boundaries

When jump table bounds checking fails, function boundaries must be established in order to reasonably limit the number of indirect branch targets. Some rewriting infrastructures do not attempt to reconstitute function boundaries. Others rely on symbolic information for the listing of all function offsets and lengths. Our goal is to reconstitute boundaries without symbolic information. We assume function bodies to be contiguous within the binary.

Determining the starting offset of a function is often straightforward, as it is typically revealed during recursive traversal as the target of a call instruction. In some cases, determining the function's ending offset is also straightforward. In functions containing no indirect CTI, all of the instruction bytes are easily discovered statically, and the instruction at the greatest file offset is considered to be the end of the function boundary.

In the case where a function *does* contain an indirect CTI, it can be conservatively assumed that the function boundary ends at the point at which the next function boundary begins. In some cases, this will result in a function boundary larger than the actual boundary. However, calculating a conservatively large function boundary serves only to increase the number of indirect branch targets, and does not compromise the correctness of the intermediate representation.

Function boundary analysis can be complicated by the presence of tail calls, which appear as branch instructions in conjunction with manual stack setup. These masked calls can be uncovered using platform-specific heuristics [17].

Functions may contain multiple entry points [29]. Our approach to function boundary determination treats the regions logically as separate functions (and disassembles them as such), but allows their file boundaries (for the purposes of branch target set identification) to overlap.

4. Related Work

Consequent to the advantages of binary rewriting, a number of binary rewriters, disassemblers, and link-time optimizers have been designed. Each rewriter can be characterized according to the way in which it deals with the challenges of binary rewriting. Existing tools handle this problem in various ways, but their approaches typically fall into several categories:

Analysis tools. These tools, typcially disassemblers, are useful for reverse engineering of binaries. They do not necessarily guarantee complete disassembly coverage or even correctness [16]. In some cases, the tools are not fully automated, and user input is required [9].

Minimally-invasive rewriters. These rewriters maintain the original machine code as-is to the greatest extent possible. This approach largely avoids issues related to instruction relocation, but supports only minimal transformations such as peephole optimizations and the insertion of 'trampolines' to jump into and out of instrumentation blocks [1, 2].

Assisted rewriters. These designs, sometimes referred to as link-time optimizers, rely upon access to additional information not required for execution and thus, not typically found in an executable [7, 8, 11–13, 15, 24, 28, 32–34]. Often, this includes information about symbols and link-time address calculations (relocation entries). This additional information allows the rewriter to determine the targets of indirect control transfer instructions, the establishment of function boundaries, and the disambiguation of code from data.

Dynamic rewriters. These rewriters perform analysis and transformation at runtime [1, 4, 6, 22, 25, 26, 30]. As such, they are able to overcome the usual rewriting challenges by delaying decisions until runtime, at which point the information required becomes clearly evident. As a benefit, these types of rewriters need not rely upon any additional information in order to successfully rewrite a binary. Unfortunately, these rewriters tend to suffer from significant runtime overhead, especially when performing complex transformations. For example, some dynamic rewriters intercept the application's execution at most indirect control transfers in order to ensure that the target has been rewritten. As a result, even without applying any transformations, reported overheads range from 20% for DynamoRIO [6] to 54% for PIN [22]. Similar to our approach, these rewriters perform indirect CTI translation at the usage site. However, because disassembly is performed at runtime, these rewriters need not identify all possible indirect targets prior to execution. A variety of implementations have focused on performing dynamic CTI translation within the context of a dynamic rewriter, as the rate of indirect CTI execution has been strongly correlated to the overall overhead of rewriting [18].

Despite its tremendous potential, binary rewriting technology has yet to gain widespread use. This is perhaps due to the respective limitations of the currently available approaches: no guarantee of correctness, limited transformation potential, a requirement for additional information, and prohibitive runtime overheads.

5. Implementation

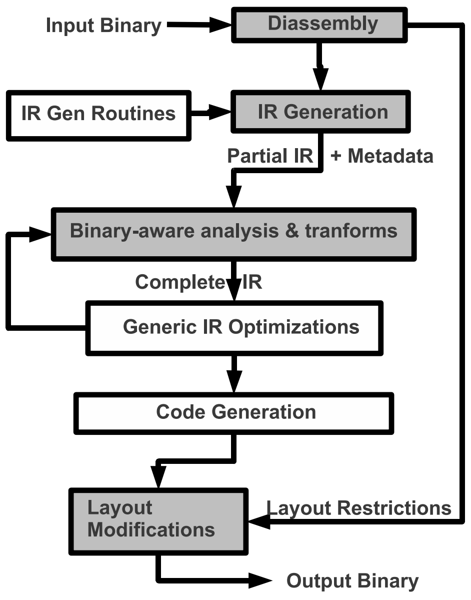

Figure 1. SecondWrite Design

The technologies presented herein have been implemented in a binary rewriter known as SecondWrite. SecondWrite integrates binary rewriting technology with the LLVM [21] compiler by disassembling input binaries into LLVM's intermediate representation (IR). As shown in Figure 1, the original binary is disassembled into an incomplete IR along with supporting metadata. This result is passed through a series of generic and binary aware analyses and transformations, such as alias analysis, constant propagation, and usage point translator refinement (target set reduction). The resultant IR is subsequently sent through code generation, where the IR undergoes a process of (re)compilation, including instruction selection and register allocation. Finally, the rewritten object code, along with specific layout restrictions (specifying where the orginal copies of the code and data segments should be located in the rewitten binary) are provided to the linker in order to produce the output binary.

Figure 2 shows a simplified overview of an example rewritten binary. The top portion of the figure illustrates that the original code and data segments are retained in the output binary. The rewritten portion of the binary contains a variety of function types. Function A represents those functions not containing any indirect branches (and thus no speculative sequences). Function A *does* contain an indirect call,

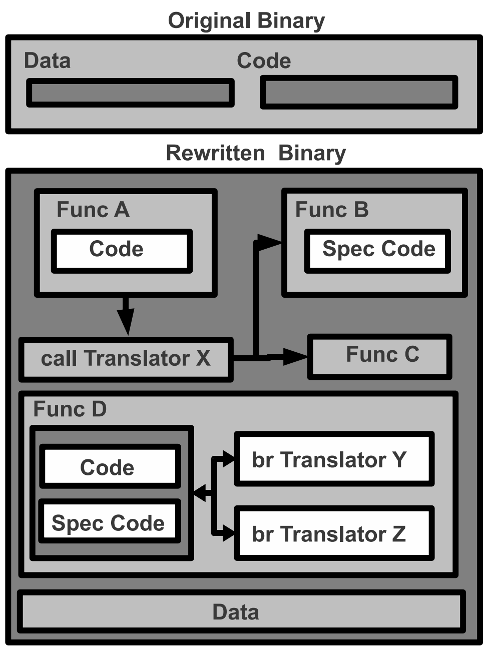

Figure 2. Example of a Rewritten Binary

which is illustrated by the edge to translator X, which redirects control flow to either B or C. Function B contains only speculative code, and is therefore considered to be a *speculative function*. Function D illustrates the possibility of a regular function containing speculative sequences and their associated branch translators. Finally, the rewritten binary itself reserves the right to introduce new data into the data segment.

6. Results

We gathered results from a subset of SPEC 2006 benchmarks: bzip2, equake [6], hmmer, lbm, libquantum, mcf, and sjeng, the largest of which is hmmer (35,992 lines of C code). Each benchmark was compiled with gcc-4.3 at optimization levels 0 and 3, to produce 14 input binaries. Each rewritten binary was verified for correctness and tested for performance via the associated SPEC dataset.

benchmark	total	indir branch	indir call
bzip2	20,900	2 (0.01%)	20 (0.10%)
equake	6738	0 (0.00%)	0 (0.00%)
hmmer	85,543	28 (0.03%)	9 (0.01%)
lbm	7,415	0 (0.00%)	0 (0.00%)
libquantum	13,579	0 (0.00%)	0 (0.00%)
mcf	3,159	0 (0.00%)	0 (0.00%)
sjeng	31,126	16 (0.05%)	1 (0.00%)

Figure 3. Number of total, indirect branch and indirect call instructions in non-optimized (O0) benchmarks.

[6] equake was taken from the SPEC OMP 2001 benchmark suite

benchmark	total	indir branch	indir call
bzip2	14,542	2 (0.01%)	20 (0.14%)
equake	7,316	0 (0.00%)	0 (0.00%)
hmmer	72,198	21 (0.03%)	9 (0.01%)
lbm	3,250	0 (0.00%)	0 (0.00%)
libquantum	12,274	0 (0.00%)	0 (0.00%)
mcf	3,124	0 (0.00%)	0 (0.00%)
sjeng	28,569	14 (0.05%)	1 (0.00%)

Figure 4. Number of total, indirect branch and indirect call instructions in optimized (O3) benchmarks.

We examined each binary for the prevalence of indirect CTI. Figure 3 displays the number of instructions for each input benchmark, when compiled without optimization. Also shown are the number of indirect branches and calls, as well as their respective percentage of the overall instruction count. Figure 4 displays the same information for the benchmarks when compiled with optimization.

These figures serve to first illustrate that although SecondWrite is still an early prototype, it is currently able to rewrite binaries with sizable instruction counts. Secondly, these figures highlight the relatively small percentage of indirect CTI that are found in C-based programs. Notably, only bzip2, hmmer, and sjeng actually contained any indirect CTIs. Future work will examine the effect of our approach on benchmarks with a larger percentage of indirect CTI, such as those compiled containing virtual tables (C++), and exception handlers.

benchmark	O0 total (reduced)	O3 total (reduced)
bzip2	124 (3)	83 (3)
equake	42 (0)	39 (0)
hmmer	601 (23)	549 (23)
lbm	36 (0)	33 (0)
libquantum	143 (3)	125 (3)
mcf	42 (0)	41 (0)
sjeng	180 (9)	164 (9)

Figure 5. Function Counts and Possible Indirect CTI Targets (Binary Characterization Target Set Reduction)

Figure 5 shows the number of functions discovered in each benchmark along with the number identified by binary characterization as possible targets of indirect CTI. Note that binary characterization is quite effective in reducing the size of the overall target set. In fact, for those binaries containing indirect calls, binary characterization was able to reduce the target set by an average of over 96%.

Our rewriting technologies enables arbitrarily complex compiler-level transformations to be applied statically to binaries. We demonstrate this achievement by applying a suite of common optimizations via LLVM during our rewriting process. The resultant execution speeds of the rewritten,

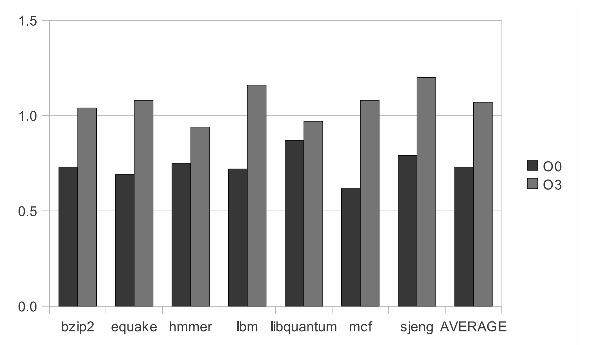

Figure 6. Runtime vs. Original

(re)optimized binaries versus the original input is shown in figure 6, with lower numbers indicating better performance.

Note that we drastically optimize the set of O0 binaries, resulting in an average speedup of 27%. SecondWrite was also able to largely maintain the performance of the optimized input benchmarks as well, producing an average slowdown of only 7%. The slowdowns experienced by some rewritten binaries are largely related to current limitations SecondWrite's static analysis capabilities. In particular, algorithms to identify function arguments and promote register spills are still very conservative.

Our recompilation technology strives to serve as a platform to enable end-user optimizations and transformations. As such, it does not aim to be optimization technology in and of itself, but merely to provide an extremely versatile platform with non-prohibitive overheads upon which to apply compiler-level transformations. *To that end, other researchers have leveraged this technology to implement automatic parallelization on binaries, realizing an average speedup of 5.1 for a suite of dense-matrix programs when transforming a serial binary for execution on an 8-core machine [20].*

7. Conclusion

We have presented a comprehensive scheme for disassembly, analysis, and layout, leveraging existing and newly developed technologies, that overcomes the limitations of current rewriters. By removing the requirement for relocation entries, this technology may serve as a future platform for providing complex transformational capabilities to end-users at reasonable overheads.

References

[1] *Dynamic Program Instrumentation for Scalable Performance Tools*, May 1994. Scalable High Performance Computing Conference.

[2] *Instrumentation and Optimization of Win32/Intel Executables Using Etch*, August 1997. USENIX Windows NT Workshop.

[3] A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2, Windows XP

Tablet PC Edition 2005, and Windows Server 2003. Technical Report 875352, Microsoft Corporation, September 2006. http://support.microsoft.com/kb/875352.

[4] *DynInst Programmer's Guide, Release 5.1*, March 2007. URL `www.dyninst.org`.

[5] M. Abadi, M. Budiu, Úlfar Erlingsson, and J. Ligatti. Control-flow integrity. In *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-226-7. doi: http://doi.acm.org/10.1145/1102120.1102165.

[6] D. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, 2004.

[7] R. Cohn, D. Goodwin, P. G. Lowney, and N. Rubin. Spike: an optimizer for alpha/nt executables. In *NT'97: Proceedings of the USENIX Windows NT Workshop on The USENIX Windows NT Workshop 1997*, pages 17–24, Berkeley, CA, USA, 1997. USENIX Association.

[8] R. S. Cohn, D. W. Goodwin, and P. G. Lowney. Optimizing alpha executables on windows nt with spike. *Digital Tech. J.*, 9(4):3–20, 1998. ISSN 0898-901X.

[9] *IDA Pro Disassembler*. DataRescue, Belgium, 2007.

[10] B. De Sutter, B. De Bus, K. De Bosschere, P. Keyngnaert, and B. Demoen. On the static analysis of indirect control transfers in binaries. In H. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, volume 2, pages 1013–1019, Las Vegas, 6 2000. CSREA Press.

[11] B. De Sutter, B. De Bus, and K. De Bosschere. Link-time binary rewriting techniques for program compaction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(5):882–945, September 2005.

[12] B. De Sutter, L. Van Put, D. Chanet, B. De Bus, and K. De Bosschere. Link-time compaction and optimization of ARM executables. *ACM Transactions on Embedded Computing Systems*, 6(1), February 2007.

[13] A. Edwards, H. Vo, A. Srivastava, and A. Srivastava. Vulcan binary transformation in a distributed environment. Technical report, 2001.

[14] U. Erlingsson. *The inlined reference monitor approach to security policy enforcement*. PhD thesis, 2004. Adviser-Fred B. Schneider.

[15] A. Eustace and A. Srivastava. Atom: a flexible interface for building high performance program analysis tools. In *TCON'95: Proceedings of the USENIX 1995 Technical Conference Proceedings on USENIX 1995 Technical Conference Proceedings*, pages 25–25, Berkeley, CA, USA, 1995. USENIX Association.

[16] *Objdump*. Free Software Foundation, Boston, MA, USA, 2007.

[17] L. C. Harris and B. P. Miller. Practical analysis of stripped binary code. *SIGARCH Comput. Archit. News*, 33(5):63–68, 2005. ISSN 0163-5964. doi: http://doi.acm.org/10.1145/1127577.1127590.

[18] J. D. Hiser, D. Williams, W. Hu, J. W. Davidson, J. Mars, and B. R. Childers. Evaluating indirect branch handling mechanisms in software dynamic translation systems. *Code Generation and Optimization, IEEE/ACM International Symposium on*, 0:61–73, 2007.

[19] C. M. Huneycutt, J. B. Fryman, and K. M. Mackenzie. Software caching using dynamic binary rewriting for embedded devices. In *ICPP '02: Proceedings of the 2002 International Conference on Parallel Processing (ICPP'02)*, page 621, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1677-7.

[20] A. Kotha, K. Anand, M. Smithson, G. Yellareddy, and R. Barua. Automatic parallelization in a binary rewriter. In *MICRO 43: Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, Atlanta, GA, USA, 2010. ACM.

[21] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (GCO)*, pages 75–87, 2004.

[22] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, volume 40, pages 190–200. ACM New York, NY, USA, 2005.

[23] J. Marathe, F. Mueller, T. Mohan, B. R. de Supinski, S. A. McKee, and A. Yoo. Metric: tracking down inefficiencies in the memory hierarchy via binary rewriting. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 289–300, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1913-X.

[24] R. Muth, S. K. Debray, S. A. Watterson, and K. D. Bosschere. Alto: A link-time optimizer for the compaq alpha. *Software - Practice and Experience*, 31(1):67–101, 2001.

[25] S. Nanda, W. Li, L.-C. Lam, and T. cker Chiueh. Bird: Binary interpretation using runtime disassembly. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 358–370, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2499-0. doi: http://dx.doi.org/10.1109/CGO.2006.6.

[26] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, 2007. ISSN 0362-1340.

[27] M. Prasad and T. Chiueh. A binary rewriting defense against stack based overflow attacks. In *Proceedings of the USENIX Annual Technical Conference*, San Antonio, TX, June 2003.

[28] B. Schwarz, S. Debray, and G. Andrews. Plto: A link-time optimizer for the intel ia-32 architecture. In *Proc. 2001 Workshop on Binary Translation (WBT-2001)*, Sept. 2001.

[29] B. W. Schwarz. *Post Link-Time Optimization on the Intel IA-32 Architecture*. PhD thesis, University of Arizona, Tucson, AZ, May 2002.

[30] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. Davidson, and M. Soffa. Retargetable and reconfigurable software dynamic translation. In *Proceedings of the international sympo-*

sium on Code generation and optimization: feedback-directed and runtime optimization, pages 36–47. IEEE Computer Society Washington, DC, USA, 2003.

[31] E. G. Sirer, R. Grimm, A. J. Gregory, and B. N. Bershad. Design and implementation of a distributed virtual machine for networked computers. *SIGOPS Oper. Syst. Rev.*, 33(5):202–216, 1999. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/319344.319165.

[32] A. Srivastava and D. W. Wall. OM: A practical system for intermodule code optimization at link-time. *Journal of Programming Languages*, 1(1):1–18, December 1992.

[33] B. D. Sutter, S. Debray, , and B. D. Bus. *Squeeze 0.3.4-Ghent for Tru64Unix User's Manual*. URL `http://www.cs.arizona.edu/projects/squeeze/`.

[34] L. Van Put, D. Chanet, B. De Bus, B. De Sutter, and K. De Bosschere. Diablo: a reliable, retargetable and extensible link-time rewriting framework. In *Proceedings of the 2005 IEEE International Symposium On Signal Processing And Information Technology*, pages 7–12, Athens, December 2005. IEEE.