Static Binary Rewriting without Supplemental Information

Overcoming the Tradeoff between Coverage and Correctness

Matthew Smithson, Khaled ElWazeer, Kapil Anand, Aparna Kotha, Rajeev Barua
Department of Electrical and Computer Engineering
University of Maryland, College Park, USA
{msmithso,wazeer,kapil,akotha,barua}@umd.edu

***Abstract*—Binary rewriting is the process of transforming executables by maintaining the original binary's functionality, while improving it in one or more metrics, such as energy use, memory use, security, or reliability. Although several technologies for rewriting binaries exist, static rewriting allows for arbitrarily complex transformations to be performed. Other technologies, such as dynamic or minimally-invasive rewriting, are limited in their transformation ability.**

We have designed the first static binary rewriter that guarantees 100% code coverage without the need for relocation or symbolic information. A key challenge in static rewriting is content classification (*i.e.* deciding what portion of the code segment is code versus data). Our contributions are (i) handling portions of the code segment with uncertain classification by using speculative disassembly in case it was code, and retaining the original binary in case it was data; (ii) drastically limiting the number of possible speculative sequences using a new technique called binary characterization; and (iii) avoiding the need for relocation or symbolic information by using call translation at usage points of code pointers (i.e. indirect control transfers), rather than changing addresses at address creation points. Extensive evaluation using stripped binaries for the entire SPEC 2006 benchmark suite (with over 1.9 million lines of code) demonstrates the robustness of the scheme.

I. Introduction

Reverse engineering binary executable code is commonplace today, especially for vulnerable code, untrusted code, and malware. Agencies as diverse as anti-virus companies, security consultants, code forensics consultants, and law-enforcement agencies routinely attempt to understand and/or rewrite binary code. Binary code is often examined because either (i) the source code is not available (such as for IP-protected commercial code, third-party code, or malware); (ii) the source code has been lost; (iii) the end-user of the binary rewriter does not trust the developer, and wants independent verification of the code; or (iv) the compiler does not offer the security guarantees needed. If capable binary rewriters become available, we envisage growth in the areas of analyzing third-party binary code for vulnerabilities [1], and rewriting the code to plug those vulnerabilities. We also anticipate increasing use in the area of enforcing security via added checks in third-party code [2, 3], including potentially untrusted code. Outside of security, a potential use of binary rewriting exists for recovering source code from legacy software. It is a common occurrence for legacy software, which is often decades old, to have the source code become unavailable. This happens because of a variety of reasons including poor record-keeping, employee turnover and retirement, corporate mergers and re-structuring, and loss of records. Sometimes even if the source code is available, it becomes unusable because it relies on an old build environment which cannot be re-created today.

To a lesser extent, binary rewriting has also been proposed for program optimization. Efforts in using binary rewriting for optimization include inter-procedural optimization [4, 5], cache optimization [6], program parallelization [7], software caching [8], and distributed virtual machines for networked computers [9].

Existing binary rewriting frameworks have failed to deliver a platform that is capable of meeting the full vision of binary rewriting as described above. This includes both static rewriters, which rewrite an executable into another executable without running it; and dynamic rewriters, which rewrite binaries during their execution into a temporary code cache. To understand the shortcomings of existing rewriters, we note that in order to perform the various tasks above, a binary rewriting framework must satisfy at least the following four criteria:

- **Complete code coverage** A good rewriter should ensure that 100% of the code can be rewritten if desired. The fraction of the code that they can rewrite is called the *code coverage*. Covering all the code is important especially for security applications where non-rewritten code may hide harmful behaviors that cannot be analyzed or corrected by the rewriter. It is also important for legacy code, where the recovered source code for the entire application is required. It is also needed for code optimization, where unknown side effects in non-rewritten code may inhibit optimizations.
- **Guaranteed correctness** For obvious reasons, the code produced by the binary rewriter should execute correctly.
- **Analysis and transformation ability** The rewriter should be capable of statically analyzing and transforming the code as needed, just like in a compiler. Deep static analysis and transformations require the precise and correct recovery of all the code, a relatively high-level intermediate representation of the binary code amenable to analysis, and adequate time to do the analysis. Indeed deep analysis is needed just to discover many source-level (high-level) artifacts from the

binary, such as variables, types, functions, arguments, and return values. Deep analysis is needed in many security applications to discover program vulnerabilities; perform information flow tracking; and reason about the behavior of untrusted code. Deep analysis is also helpful for certain program optimizations since they require knowledge of source-level artifacts. Further, recovering source-level artifacts from binaries requires complex transformations such as type recovery, variable detection, and argument and return values detection. Transformations are also required for many security schemes and optimizations.

- **Should work on stripped binaries** The rewriter should be able to rewrite stripped binaries, *i.e.* those without relocation or symbolic information. Static relocation entries are processed and then (unless explicitly instructed otherwise) discarded by the linker. As a result, almost all production binaries do not contain this information.

Surprisingly, the above four criteria are not all simultaneously met by any single rewriter today. In particular, existing static rewriters cannot ensure complete code coverage; often make unsafe assumptions that may break correctness; and cannot rewrite stripped binaries. Dynamic rewriters also do not ensure 100% code coverage; only have the time for the most rudimentary forms of program analysis and transformations; produce no durable output code; and most analysis conclusions they do reach are generally input-data-dependent.

To understand the drawbacks of existing rewriters, it is helpful to understand a key problem in binary rewriting: distinguishing code from data. We call this the *content classification* problem. Content classification is difficult because binary executables contain not code, but also data. This includes data which may be buried inside code sections at any location where control does not flow to. Examples of such data include literal constants, jump tables, and literal tables. For coverage, rewriters must recognize and rewrite *all* of the code; however for correctness they must rewrite *none* of the data mistakenly as code, since data must be preserved.

In the following sections, we present the underlying technologies of our binary rewriter known as SecondWrite. SecondWrite is the first static binary rewriter that guarantees 100% code coverage without the need for relocation or symbolic information. Our contributions are (i) a speculative approach for handling portions of the code segment with uncertain content classification; (ii) drastically limiting the number of speculative sequences by introducing a technique known as binary characterization; and (iii) avoiding the need for relocation or symbolic information by using address translation at usage points of code pointers (*i.e.* indirect control transfers), rather than changing addresses at creation points.

This paper differs from previous publications related to SecondWrite. Previous publications focused on automatic parallelization [7], symbol promotion [10], variable and function argument recognition [11], and security check insertion [3]. None of these publications deal with the underlying problem of disassembling and rewriting binaries without relocation information, the main topic of this paper. However, before presenting our solution, we first look at existing rewriting approaches in more detail, in light of the criteria above.

A. Dynamic Rewriting

Dynamic rewriting technologies can be found in rewriters such as Pin [12], Bird [13], and DynInst [14]. These tools perform disassembly, analysis, and transformation while the target program is executing. The code is rewritten to a temporary cache in binary form. No durable code is output. The benefit of dynamic rewriting is that content classification (*i.e.* distinguishing code from data) is delayed until runtime, where it is easy since only code that is actually executed is rewritten. 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.

The main drawback of dynamic rewriters is that they do not have the time to run complex tasks required by many analysis techniques since the analysis time gets added to the application run-time. Binary rewriters require many complex analyses to recover source features and analyze the program; such required analyses include type recovery, symbol analysis, alias analysis, and information flow tracking. As a result of this drawback, uses of dynamic rewriters has been restricted to simple tasks such as code instrumentation and peep-hole optimization, since they cannot afford to run more complex tasks.

A second drawback is that dynamic rewriting often has high overheads even without complex analyses. For example, reported overheads range from 20% for DynamoRIO [15] to 54% for Pin [12]. A third drawback of dynamic rewriters is that they only rewrite the code that is reachable with the given input data set, and therefore any deductions they make are only valid for that input data. Hence they are limited in their ability to draw any general conclusions about the program's behavior.

B. Static Rewriting

Static binary rewriting technology can be found in rewriters, disassemblers, and link-time optimizers such as IDA Pro Disassembler [16], objdump [17], OM [18], Atom [19], Spike [20], Diablo [21], Alto [5], and PLTO [4]. These technologies are designed for rewriting code offline. Given their offline nature, they have the time to perform complex analysis and transformations.

A major challenge with static rewriting is content classification (also known as distinguishing code from data), which other research has determined to be 'unsolvable' [22] statically. To understand why, consider that there may be data buried inside code sections. Hence static techniques are used to discover what portions of the binary program are code so that they can be rewritten, while preserving the data without modification. In general the only way to prove that a sequence of bytes in the program is code is to discover a guaranteed control-flow path from the program's entry point to that sequence of bytes. This method is called *recursive*

traversal. This is statically easy to do only via direct control-flow transfer instructions (CTIs). For indirect CTIs, *i.e.* those whose address is decided at run-time, it is hard to prove which addresses may be chosen, and hard to prove that any possible control-flow path will actually be taken with some input data set. Without these guarantees, many code targets cannot be proven, leading to incomplete code coverage.

Another problem with static rewriters is that all instruction locations may move after rewriting. Handling such instruction movement means stripped binaries cannot be rewritten, since existing static schemes use relocation or symbolic information. To see why, consider that with instruction movement, all indirect control-transfer instructions (CTIs) must jump to moved addresses. To do this all existing static rewriters rely on updating all address creation points (ACPs) in the binary, which are the points the addresses of indirect CTIs are created. However, the location of ACPs is not apparent, and requires relocation or symbolic information to derive a list. As a result, stripped binaries cannot be rewritten.

A third drawback of static rewriting is that they often make unsafe assumptions that may violate correctness. For example, Harris et al [23] assumes that instruction sequences that appear to be function prologues *are*, in fact, function prologues. This is an unsafe assumption since a portion of the binary file that is not a function (such as data) may nevertheless coincidentally have a sequence of bytes that looks like a function prologue. If this happens, the output code of the rewriter will be wrong. Another common behavior is that jump tables are recognized using compiler-specific patterns in the binary code. However these heuristics may fail, leading to incomplete coverage or wrong output, depending upon subsequent handling.

C. Minimally-Invasive Rewriting

Minimally-invasive rewriting, found in rewriters such as Etch [24], is a static approach that works on stripped binaries, but sacrifices coverage and the ability to perform complex transformations. In this approach, the binary image is kept mostly unchanged; as a result all targets of indirect control transfers remain in place. Hence no ACP translation is needed, so stripped binaries work. However they still suffer from the same problem of incomplete code coverage that all other static rewriters have. Moreover the requirement to keep the code mostly unchanged greatly hampers the ability to apply transformations. As a result only minimal transformations are allowed, such as the insertion of 'trampolines' to jump into and out of instrumentation blocks [14], and peephole optimizations affecting small sequences of instructions.

D. Our New Approach

We present a new static binary rewriting technology that can support arbitrarily complex transformations, provides 100% code coverage, makes no unsafe assumptions, and works for stripped binaries. This allows rewriting to achieve its full potential. The key idea behind our approach is avoid the unsolvable problem of content classification by treating portions within the code segment that cannot be definitively classified as both code *and* data. Unclassified portions are disassembled as speculative code. They are also copied, unmodified into the output binary, and loaded to their original memory locations, so that any data references to the unclassified portions will continue to work. The need for relocation information is avoided by not updating ACPs; instead, indirect CTIs are translated at run-time using a light-weight address translator.

Our technologies are applicable to any static rewriter. However our evaluation platform is SecondWrite, which relies on other technologies we have developed, such as those to convert binaries to a high-level intermediate representation (IR) [10, 11]. High-level IR is helpful for allowing effective static analysis and transformations of the code. However it is not essential for the functioning of the method in this paper.

We recognize that rewriting all programs statically is a very challenging problem. This work should be seen as what it is: the first successful attempt to build a static rewriter which can rewrite stripped binaries while guaranteeing both correctness *and* complete code coverage. Statically handling every program in the world may still be an elusive goal. Legitimate binaries (which are the main target of this work) are easier to rewrite, but regarding malwares, some are obfuscated or packed; unpacking tools such as Quick unpack [25] may need to be run in advance. We expect future work incorporating dynamic feedback to the static rewriter will be helpful in such cases. Nevertheless, we believe that expanding the scope and practicality of static rewriting in this paper is a valuable contribution to the community.

II. Relocating Instructions

Let us consider how static binary rewriters account for instructions or data that move upon rewriting. Load and store instructions reference locations containing data. Branches and calls reference locations containing other instructions. Rewriters must first identify the places in the binary where constant target addresses are stored, 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, 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* (*i.e.* the indirect call). Thus, the identification of ACPs for indirect CTI is non-trivial.

To overcome the problem of ACP identification, 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 to account for all indirect CTI targets. Thus, to account for instruction 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. Unlike traditional static rewriters, we handle indirect address translation by updating the address *usage* point instead of the address *creation* point. This solution, applying usage point translation to a static rewriting technology, is a central novelty to our scheme.

Before expanding upon our approach, let us first consider the alternatives. 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:

```
0x8200: mov $0x8900, %eax
```

Assume that in the original binary, address 0x8900 corresponds to the base address of function foo, and that function foo was subsequently rewritten to a different location. It would be unsafe to modify the above move instruction's operand to point to the new location of foo, unless we can somehow prove that the operand actually represents an address rather than a data value 0x8900 that coincidentally looks like an address. Suppose that we did choose to modify the operand, but the operand was actually representative of a data value (perhaps a loop bound). In this case, the rewritten program would be incorrect.

On the other hand, if we choose *not* to update the operand, but the operand actually *did* represent the address of foo (an indirect call operand), then the program will also be incorrect. In this case, 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.

Unlike static ACP translation, our usage point translations approach avoids the requirement of definitively identifying 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 adjust 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. Since no translation is done for data values, they remain unchanged.

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 where the rewriter is able to statically prove that fp can only point to foo (located at address 0x8300 in the original binary) or bar (located at address 0x8400). Larger target sets are accommodated by simply adding additional cases. In this example, the symbols foo and bar represent the created function symbols in the rewritten IR.

```
if (fp == 0x8300):
    fp_modified = &foo;
else if (fp == 0x8400):
    fp_modified = &bar;
else:
    assert(false);
call *fp_modified;
```

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 the 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 foo is not an actual target for fp. In this case, including foo in the translator is useless, as that particular translation will never be executed. *However, the presence of the foo 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 location in the code section*. In this way, we guarantee that we will always rewrite 100% of the binary's code.

Clearly, any usage point translation will introduce run-time overhead. Indeed the translation code as shown above with cascading if-else statements will have high overheads when the number of possible targets is large. There we do *not* use the code structure above.

Instead, we store translated addresses in a table which we can access with O(1) overhead, regardless of the number of targets. Our table contains an entry, for each byte offset within the code section. Each entry stores the translated address for the associated byte offset. For the x86 ISA, which uses 32-bit addresses, each table entry requires 4 bytes of space, yielding a translation table that is 4 times the size of the original binary's text section. Later, we will present techniques for eliminating a vast number of possible indirect CTI targets. This has the effect of producing a large number of unused (empty) entries in the translation table. In this case, a more dense hash table structure could be used to store the translated addresses, at the potential expense of additional runtime overhead in order to check for collisions.

III. Relocating Data

The previous section discusses how instruction references are adjusted to account for movement during rewriting. However, the solution did not address indirect data references.

Our approach maintains correctness of indirect data references by prohibiting movement of 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. Stack and heap segments are not part of the input binary and their addresses are run-time determined; hence there is no need to anchor these segments during rewriting.

Note that this approach leads to some duplication. The original code segment will exist in the rewritten binary alongside its functionally-equivalent rewritten copy. Our results show that despite this size increase, the runtime overhead of our scheme was measured to be negligible.

IV. Code Discovery

In addition to instruction relocation, binary rewriters must overcome the challenge of identifying which portions of the binary contain instructions. Our approach addresses this challenge by employing a technique known as speculative disassembly. Before explaining our speculative technique in more detail, let us motivate the problem and discuss the issues in more detail.

At first glance, the process of code discovery appears to be trivial. Binaries must adhere to common file formats such as the Executable Linking Format (ELF) or the Portable Executable (PE) format in order to facilitate loading by a target operating system. These file formats will typically require the program to be separated into executable code segments and non-executable data segments.

However, this simplistic view is complicated by the presence of data 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 is known as linear sweep [26, 17]. This algorithm 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 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 [26], which discovers code by following only valid control-flow edges. As CTIs are 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. Our findings show that recursive traversal alone covers less than 1% of the binary due to indirect CTI present in compiler-inserted startup code.

The previous section discussed how static rewriters use relocation entries to identify indirect CTI targets stored at address creation points (ACPs). 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, extraneous CTI targets imply 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.

To correctly handle portions of the code segment which we are not sure are code or data, we perform speculative recursive traversal disassembly on those portions as if they contained instructions, even though that is not guaranteed to be the case. Since we employ usage point translation rather than creation point translation, and retain a copy of the original code segment, our method maintains correctness as we show next.

Let us examine how our method would handle a portion of data mistakenly identified as a possible indirect CTI target. First, it will be speculatively disassembled as if it contained instructions. A translation will 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 targeted the location. Thus, in the rewritten binary, the translator will never redirect execution to the speculatively-disassembled sequence, thus maintaining correctness. As mentioned previously, we maintain a copy of the original code segment in place in order to guarantee that any data references to this region will also maintain their correctness.

Although other research has used speculative techniques for code discovery in order to increase code coverage [27], our method is the first to incorporate speculation with usage-point translation in order to guarantee both 100% disassembly coverage while also maintaining correctness.

The speculative disassembly process can help to reduce the target set size for a given indirect CTI 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.

We prune invalid code sequences from the IR. On average, we eliminate only 3.7% of speculative code sequences using

this technique. The number that can be pruned is low, as the x86 ISA has few invalid opcodes, and only a small portion of the binary is categorized as 'known code' due to the presence of indirect CTI within startup code. The following sections will discuss additional mechanisms for reducing the target set size for a given indirect CTI.

V. Optimizing Target Sets

Sections II and IV presented an approach for discovering and relocating code statically and without supplemental information. Although the technology guaranteed correctness of the rewritten binary, it is extremely conservative in identifying indirect CTI targets. This section identifies the downsides to this conservative approach, and presents some solutions, the most powerful of which is a new technique known as binary characterization.

Without further optimization of the speculative target set, our technique would be forced to accept every offset within every gap as a potential starting point for a speculative function. Restarting disassembling at every offset is a technique referred to as 'speculative completion' [27]. Extraneous targets increase code size, and also increase the complexity of the intermediate representation by introducing unnecessary control flow edges, which can hinder inter-procedural transformations. Most importantly, introducing too many extraneous targets can lead to very large IR, prohibiting the ability of our scheme to realistically scale to large binaries. To address these concerns, we can apply the following techniques for reducing the size of indirect CTI target sets through the elimination of false targets.

A. Constant Propagation

Constant propagation is a dataflow optimization where the use of a variable assigned to only a single constant is replaced by that constant. Indirect call target identification via constant propagation is a technique used by DeSutter et. al. [26]. It was discovered that the targets of 92% of indirect calls could be discovered via propagation. However, these particular results are heavily reliant upon the Alpha architecture, where all inter-modular 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 does not apply.

B. Binary Characterization

One key novelty to our entire scheme is 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 prove that a location is *not* an address. Thus, it is possible to generate a reduced 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 contents of the code and data segments are subsequently scanned for binary patterns that could represent addresses within the constructed range, taking into account the endian-ness 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. Our results show that binary characterization alone eliminates 99.88% of all speculative targets.

Binary characterization is only appropriate for reducing target sets in those situations where addresses are not calculated at runtime. This will be explored further (see section VIII).

C. 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.

Alias analysis is a dataflow analysis that can identify which locations can be pointed to by each program address (such as a CTI operand). Alias analysis can provide information about which ACP may be contained within a CTI operand even if the operand is passed through global memory locations, the stack, function arguments, or registers. We leverage the results of Andersen's algorithm [28] for 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 CTI operand may alias a set of targets, those entries not included in the 'may alias' set can be eliminated from the indirect CTI's translator.

Further optimization is possible in special 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. In situations where a set of targets is known to 'must alias' an operand, traditional creation point translation can be used in favor of usage point translation.

VI. Function Boundaries

Although the correctness of our scheme is *not* reliant upon reconstitution of the function boundaries present in the original binary, our disassembly process does split the IR into individual functions at the following locations:

- The targets of direct CALL instructions
- Binary characterization entry points
- BRANCH targets where a function lies between the instruction and the target

Note that because indirect branch targets will appear as binary characterization entry points, jump table targets will be extracted into separate functions. Although this does create IR that appears different from the original, it remains functionally equivalent. If desired, heuristics that identify jump tables [29] can help recover function boundaries more representative of the original binary.

VII. Callbacks

So far, we have used usage-point translation of target addresses at indirect CTIs to account for instruction movement during rewriting. However this is not possible for callbacks. A *callback* happens when a function pointer is passed as an argument to an external library function. Later the library function calls the pointed-to function in the application, which is the callback event. This section presents an optimized solution for handling most callbacks (static ACP translation), as well as fall-back solutions to handle the remaining cases.

Let us first understand why callbacks present a problem to static binary rewriters. Because callbacks to application functions are executed outside of the application being rewritten, a static rewriter cannot translate the address at the point of call. Without modification, this address will refer to the function's address in the original binary, not the rewritten binary, leading to incorrect execution.

There are two possible solutions to the problem of untranslated callback function addresses. In the first solution, we identify callbacks by finding function pointer arguments passed to external library calls. However, identifying argument types requires prototype information to be available, which is not generally available within in the binary itself. We use a solution where we have collected the prototypes of all standard libraries, such as C run-time libraries and Win32 API libraries. Once the callback arguments have been identified, our tests show that on average, 82% of callback arguments are constants, allowing SecondWrite to perform static translation. For the remaining non-constant arguments, instructions are inserted to translate the callback address just prior to making the library call.

However, in the rare situation in which prototype information is not available for a particular library function

, it is unsafe to modify call arguments. When untranslated functions perform callbacks, they will target the original code segment. SecondWrite avoids execution within the original code segment by marking the segment as non-executable. During execution, any callback to the original code causes the operating system to produce a segmentation fault. To ensure correctness, we register a custom segmentation handler during startup which translates all original code addresses to rewritten code addresses using our translator and then restores control flow to the proper location within the rewritten code segment.

VIII. Limitations

Thus far, we have presented an approach for performing static binary rewriting without relocation information. This approach guarantees correctness of any rewritten binary, aside from three limitations. First, like most static binary rewriters, self-modifying code is not handled. Existing methods [30] could be integrated to statically detect the presence of runtime code generation and prevent rewriting.

Second, our technology does not handle binaries containing obfuscated control flow. This technology relies upon the recursive traversal discovery algorithm, and assumes that CTIs exhibit normal behavior. For example, it is assumed that return addresses are not modified after being pushed onto the stack by a CALL instruction.

Additionally, our approach does not address control flow introduced as a result of software exceptions. Typically, compilers will store information about code execution within the binary in order to facilitate stack unwinding at runtime. Additional analyses, such as the examination of compiler-specific exception handling data, are necessary to maintain exception handling support in a static rewriter. Although we do not address exception handling, our approach does not prevent these analyses from being incorporated.

Finally, binary characterization, introduced below, assumes that absolute addresses will appear statically within the binary, and will not be assembled at runtime. There are two cases where runtime-computed addresses may be found in binaries.

A. Position Independent Code (PIC)

Shared system libraries may be compiled in a position-independent fashion in order to allow applications to map shared code to different base virtual memory addresses. In situations where local functions and data are indirectly referenced within position-independent code, the target address is computed at runtime. We have begun drafting a solution that will detect and track address-computing PIC instructions. Our solution will be investigated in future work.

B. Certain Jump Tables

Compilers use jump tables to implement dense switch statements. The typical implementation is to store the address of each case's code entry point in a table. The input value is used as an index into the table of absolute addresses. Although the address used to access the table is calculated at runtime, it does not present a problem, as this is a data reference. Importantly, the indirect control flow target addresses are all statically calculated.

However, an alternate implementation is possible where the basic block targets themselves (and not their addresses) are arranged in the form of a table. In this instance, an address constructed at runtime is used as an indirect CTI operand, and the absolute target addresses do *not* appear in the binary. This is a much more complex approach to implement, and only serves to eliminate one indirect reference. This approach would require the code for each case to be compiled and measured for size prior to constructing the address calculation.

Nevertheless, such an implementation is possible, and would require introduction of a heuristic to support within our scheme.

IX. Implementation

The technologies presented herein have been implemented as the basis for the disassembly within our binary rewriter known as SecondWrite. SecondWrite integrates binary rewriting technology with the LLVM [31] compiler by disassembling input binaries into LLVM's intermediate representation (IR).

The original binary is disassembled into an IR along with supporting metadata. Next the IR is passed through a series of generic and binary aware analyses and transformations, such as alias analysis, constant propagation, data set analysis, and usage point translator refinement (target set reduction). The resultant IR is sent through code generation, where it undergoes a process of (re)compilation, including instruction selection and register allocation. Finally, the rewritten object code, along with memory placement restrictions (specifying locations for the original copies of the code and data segments) are provided to the linker in order to produce the output binary.

Although SecondWrite converts the input binary to high-level IR, our scheme does not rely on such a conversion. Our method can be applied equally to rewriters such as PLTO [4], which maintain a lower-level representation of the binary throughout the rewriting process.

Figure 1 shows a simplified overview of an example rewritten binary. The figure shows the original code and data segments retained in the output binary. The rewritten binary contains a combination of functions guaranteed to be code (A and C), and speculative functions (B). The binary also contains a callback and its associated stub that serves to preserve the original function prototype. Function A contains an indirect call, which is illustrated by the edge to the call translator, which then redirects control flow to either B or C.

X. Results

We tested SecondWrite on the SPEC CPU2006 benchmark suite, which includes 29 benchmarks written in C, Fortran, C++ or some combination thereof. Each input benchmark was compiled with gcc and then stripped of all symbolic information and static relocation entries.

To test the main purpose of our scheme, correctness without relocation information, each rewritten binary was tested for correctness by comparing its behavior to that of the original binary using the associated dataset. Test results indicated that SecondWrite produced correctly functioning binaries, successfully rewriting the entire SPEC 2006 benchmark suite.

Our scheme produces successful results not just for trivial binaries, but for binaries of significant size and complexity, including gcc, xalancbmk, and gamess, each of which are compiled from more than 200,000 lines of source code. In total, the binaries tested collectively were compiled from 1.9 million lines of code as measured by SLOCCount.

Areas of each benchmark not guaranteed to be code were measured. This is the area of the code segment (the gaps) that

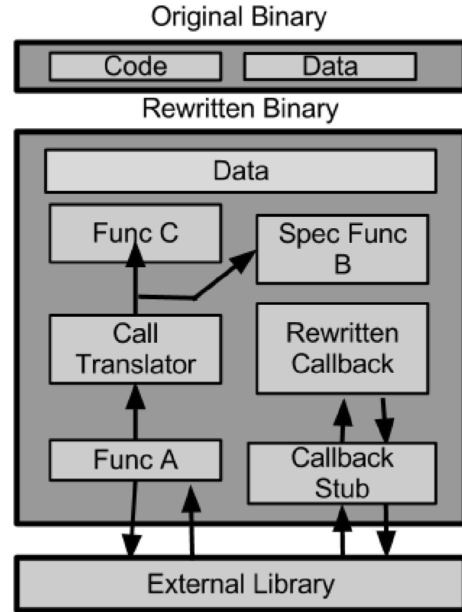

Fig. 1: Example Rewritten Binary Layout

remains after traditional disassembly. On average, more than 99% of each benchmark remained after traditional recursive traversal disassembly. This is due to the presence of indirect CTI within the startup sequence inserted into each binary during compilation. Therefore, the only code directly reachable from each binary's external entry point is a portion of the startup code, leaving the vast majority of each binary to be disassembled speculatively.

In order to calculate the effectiveness of binary characterization, we calculated the number of entry points identified by this technique. As presented earlier, without target set optimization, every offset of every gap would be a potential starting point for a speculative function. For the SPEC benchmark suite, on average binary characterization was able to successfully eliminate 99.88% of all speculative targets. Without this reduction in targets, the IR would have ballooned in size, rendering rewriting infeasible. On average, our scheme produced a 55% increase in the number of functions and a 44% increase in code size for instructions versus the input binary.

Figure 2 identifies the number of indirect CTI in each binary as discovered by SecondWrite. This includes all indirect jump and indirect call instructions encountered during disassembly, including speculative disassembly. Some binaries (typically the smaller programs), contain only a few indirect CTI while others such as xalancbmk, contain many thousands.

Figure 3 identifies, for each binary, the percentage of runtime spent performing indirect call translation as measured by Perf, a profiler tool for Linux. The overhead is broken into two portions : time spent translating the target address (address overhead) and time spent pushing memory arguments onto the stack (argument overhead). Our address overhead was found to be on average .08%. Only this first overhead is inherent to our scheme, and it would be the only overhead if our scheme were applied to most existing static binary

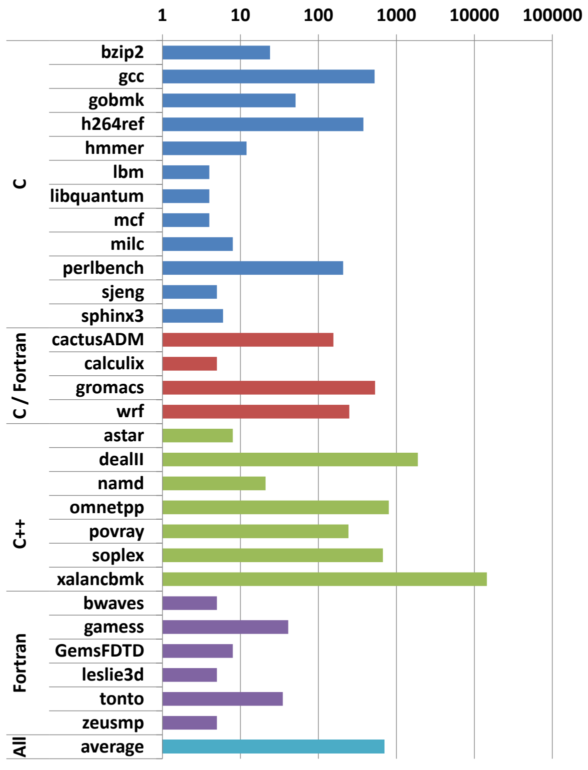

Fig. 2: Static Count for Number of CTI Instructions

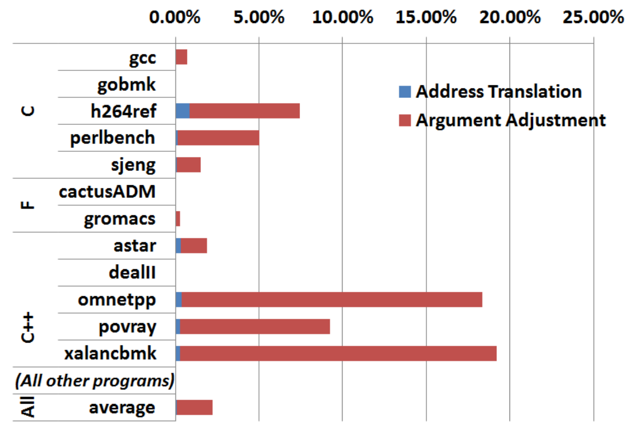

Fig. 3: Translator Runtime Overhead (as % of application runtime) Programs in Figure 2 not shown here have zero overhead.

rewriters such as PLTO [4] and ATOM [19], which represent the binary internally in a lower-level binary-like IR. This overhead just translates original addresses at each indirect CTI to their rewritten counterparts using the table lookup described in section II. The overhead is low because indirect CTIs are relatively rare, and the overhead of each translation is small (a single table lookup).

The second overhead for address translation (on average 2.12%, but higher in C++ programs which have many indirect calls) occurs only in SecondWrite since it converts binary code to a high-level source-like IR (LLVM IR in our case). SecondWrite does so because it has a more ambitious goal of not just binary rewriting, but recovery of source code from binaries. Therefore it performs register reallocation and does not maintain a monolithic view of the original stack layout from the input binary; instead SecondWrite dismantles the stack to its component source code variables [10]. When the stack is dismantled, the arguments to indirect calls must be explicitly represented in high-level code. When the output code from SecondWrite is compiled, the compiler then incurs additional overhead in copying these arguments from their original locations to the argument locations of the new functions. This argument overhead would not be present if our scheme were to be applied to more traditional rewriters that did not seek to break up the stack or reallocate registers.

Figure 4 shows the increase in the size of the rewritten binary. The rewritten binary is comprised of three different regions : rewritten code, translation table, and original content. The rewritten code segments are observed to be 44% larger on average than the original. This is expected due to the presence of speculative code. For our testing, in order to minimize callback translation runtime overhead, we chose to use the sparse lookup table, which requires a table size of 4 times the size of the original code segment.

In total, with sparse tables, the rewritten binaries were on average 6.3 times larger than the input binaries. As discussed in section IX, this code size increase can be greatly reduced by using a dense hash table. Further, we have found the code size overhead to not be a serious concern primarily because most of the code size increase only exists in disk and virtual memory, with minimal increases in physical memory and cache footprint. This is because (i) invalid rewritten speculative code is never accessed; (ii) the original code segment is never executed; and (iii) translation tables are very sparse, so only a small fraction of them are ever accessed. It is important to note that the run-time overhead (measured above as negligible) includes any run-time impact from increased code size.

Because we provided all necessary prototypes to SecondWrite for common external functions, such as the C and C++ standard library, were are able to statically translate more than 82% of all callback arguments, and insert translations for the remaining arguments. This optimization, and the fact that callbacks are rare, results in negligible overheads for callback address handling (less than .01% of the total runtime).

Researchers have leveraged our platform for many rewriting applications, such as applying symbol promotion to realize an average 8% speedup in previously-optimized binaries [10], applying automatic parallelization to realize 2.2x speedup for a subset of the *PolyBench* and *Steam* benchmark suites [7], and the insertion of security checks into untrusted binaries [3].

XI. Conclusion

The challenge for static binary rewriting is to differentiate portions of the binary that are code from portions that are data. Without supplemental information (unassisted), it is impossible to statically disambiguate code from data all of the time (complete coverage) with guaranteed certainty (perfect accuracy). Because of this, current approaches to static rewriting are limited to providing limited coverage, reduced accuracy, or

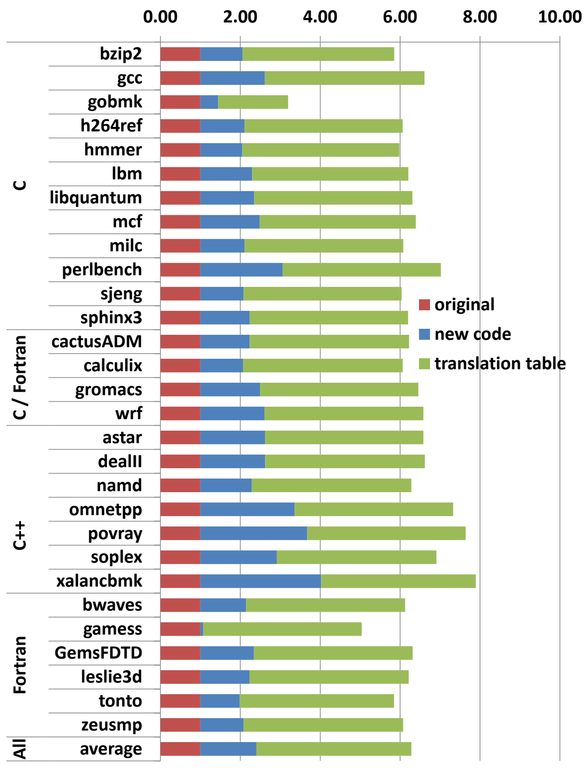

Fig. 4: Rewritten Binary Size (vs. input binary size)

requiring additional information. This paper introduces a new approach which eliminates the need to differentiate between code and data, while still maintaining correctness. We have implemented this approach as the basis for SecondWrite, the first static binary rewriter that guarantees 100% code coverage without the need for relocation or symbolic information. Our contributions are (i) handling portions of the code segment with uncertain classification by using speculation; (ii) binary characterization, a technique for limiting the number of speculative sequences; and (iii) call translation at the usage points of code pointers (instead of address creation points) to avoid the need for relocation information.

References

[1] Application Security testing - Veracode, http://www.veracode.com/.

[2] U. Erlingsson, "The inlined reference monitor approach to security policy enforcement," Ph.D. dissertation, 2004, adviser-Fred B. Schneider.

[3] P. O'Sullivan, K. Anand, A. Kotha, M. Smithson, R. Barua, and A. D. Keromytis, "Retrofitting security in cots software with binary rewriting," in *IFIP SEC*, 2011, pp. 154–172.

[4] 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.

[5] 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*, vol. 31, no. 1, pp. 67–101, 2001.

[6] 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*. Washington, DC, USA: IEEE Computer Society, 2003, pp. 289–300.

[7] A. Kotha, K. Anand, M. Smithson, G. Yellareddy, and R. Barua, "Automatic parallelization in a binary rewriter," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '43, Washington, DC, USA, 2010, pp. 547–557.

[8] 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)*. IEEE Computer Society, 2002, p. 621.

[9] 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.*, vol. 33, no. 5, pp. 202–216, 1999.

[10] K. Anand, M. Smithson, K. ElWazeer, A. Kotha, J. Gruen, N. Giles, and R. Barua, "A compiler-level intermediate representation based binary analysis and rewriting system," in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13, 2013, pp. 295–308.

[11] K. ElWazeer, K. Anand, A. Kotha, M. Smithson, and R. Barua, "Scalable variable and data type detection in a binary rewriter," in *Proceedings of the 34th ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, Seattle, WA, 2013.

[12] 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*, vol. 40, no. 6.

[13] 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*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 358–370.

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

[15] D. Bruening, "Efficient, Transparent, and Comprehensive Runtime Code Manipulation," Ph.D. dissertation, MIT, 2004.

[16] *IDA Pro Disassembler*, http://www.datarescue.com/idabase/, DataRescue, Belgium, 2007.

[17] *Documentation for binutils 2.21*, http://sourceware.org/binutils/docs-2.21/binutils, Free Software Foundation, Boston, MA, USA, 2010.

[18] A. Srivastava and D. W. Wall, "OM: A practical system for intermodule code optimization at link-time," *Journal of Programming Languages*, vol. 1, no. 1, pp. 1–18, December 1992.

[19] 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*. Berkeley, CA, USA: USENIX Association, 1995, pp. 25–25.

[20] 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*. Berkeley, CA, USA: USENIX Association, 1997, pp. 17–24.

[21] 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*. Athens: IEEE, December 2005, pp. 7–12.

[22] R. N. Horspool and N. Marovac, "An approach to the problem of detranslation of computer programs," *The Computer Journal*, vol. 23, no. 3, pp. 223–229, 1980.

[23] L. C. Harris and B. P. Miller, "Practical analysis of stripped binary code," *SIGARCH Comput. Archit. News*, vol. 33, no. 5, pp. 63–68, 2005.

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

[25] "Quick unpack." [Online]. Available: http://qunpack.ahteam.org/

[26] 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 *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, H. Arabnia, Ed., vol. 2. Las Vegas: CSREA Press, 6 2000, pp. 1013–1019.

[27] L. C. Harris and B. P. Miller, "Practical analysis of stripped binary code," *SIGARCH Comput. Archit. News*, vol. 33, no. 5, pp. 63–68, Dec. 2005. [Online]. Available: http://doi.acm.org/10.1145/1127577.1127590

[28] L. O. Andersen, "Program analysis and specialization for the c programming language," Ph.D. dissertation, University of Cophenhagen, 1994.

[29] C. Cifuentes, M. Van, E. C. Science, and E. Engineering, "Emmerik. recovery of jump table case statements from binary code," in *Science of Computer Programming*, 2001, pp. 2–3.

[30] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu, "Still: Exploit code detection via static taint and initialization analyses," in *ACSAC*. IEEE, 2008.

[31] 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)*.