Zipr: Efficient Static Binary Rewriting for Security

William H. Hawkins, Jason D. Hiser, Michele Co, Anh Nguyen-Tuong, Jack W. Davidson
Department of Computer Science, University of Virginia
{whh8b,hiser,mc2zk,an7s,jwd}@virginia.edu

***Abstract*—To quickly patch security vulnerabilities there has been keen interest in securing binaries in situ. Unfortunately, the state of the art in static binary rewriting does not allow the transformed program to be both space and time efficient. A primary limitation is that leading static rewriters require that the original copy of the code remains in the transformed binary, thereby incurring file size overhead of at least 100%.**

This paper presents Zipr, a static binary rewriter that removes this limitation and enables both space and time efficient transformation of arbitrary binaries. We describe results from applying Zipr in the DARPA Cyber Grand Challenge (CGC), the first fully automated cyber-hacking contest. The CGC rules penalized competitors for producing a patched binary whose on-disk size was 20% larger than the original, whose CPU utilization was 5% more than the original, and whose memory use was 5% more than the original. Zipr's efficiency enabled our automated system, Xandra, to apply both code diversity and control-flow integrity security techniques to secure challenge binaries provided by DARPA, resulting in Xandra having the best security score in the competition, remaining within the required space and time performance envelope, and winning a $1M cash prize.

I. Introduction

This paper describes Zipr, a novel static binary rewriting tool that builds a compact and efficient transformed binary program or library. Zipr does not require compiler support, debug information or source code. Zipr is compiler agnostic and is applicable to both dynamically- and statically-linked programs and shared and static libraries. Zipr is generally well-suited for program optimization and transformation but particularly useful in software security, reliability and dependability.

Because Zipr is applied to a program's machine code, it is complementary to static analysis, proof carrying code, etc. that protect and improve software through source code modification. User-defined transformations applied with Zipr can improve or optimize COTS and proprietary programs, even if the source code is unavailable or the developer is unable or unwilling to make changes.

Zipr is able to rewrite binary programs without keeping a duplicate of the original program's instructions, minimizing the on-disk filesize and runtime overhead of rewritten binaries. In many contexts where programs are rewritten in situ to add new functionality or security, minimizing filesize and runtime performance and memory overhead are important. For example, on embedded systems the size of the binary directly affects the economics of production [1].

Zipr was a critical component of Xandra, our entry in the 2016 DARPA Cyber Grand Challlenge (CGC) competition [2]. While CGC was structured as an autonomous hacking competition, the overall goal was to demonstrate that it was feasible to identify vulnerabilities in binary code and patch them without human intervention. A successful entrant in the competition had to "[r]everse engineer unknown software" (i.e., software without source code or other metadata) and "[h]eal weaknesses without sacrificing [f]unctionality, [or] [c]orrectness [or] [p]erformance" [2]. CGC was an excellent test scenario to demonstrate that Zipr could produce rewritten binaries that have acceptable filesize and performance overhead with respect to the original binary.

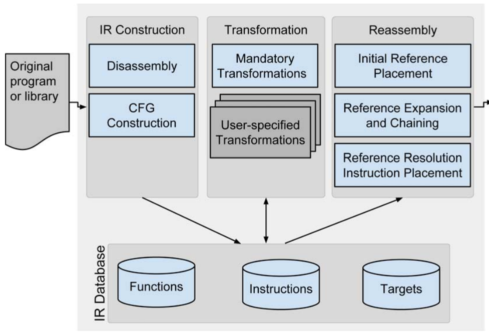
IR Construction
Transformation
Reassembly
Original program or library
Disassembly
CFG Construction
Mandatory Transformations
User-specified Transformations
Initial Reference Placement
Reference Expansion and Chaining
Reference Resolution Instruction Placement
Modified program or library
IR Database
Functions
Instructions
Targets

Fig. 1. The Zipr rewriting pipeline.

The remainder of this paper is organized as follows: Section II describes the algorithm used by Zipr for statically rewriting a binary computer program, while Section III describes how Zipr optimizes code layout. Section IV presents the results of an evaluation of Zipr by describing its ability to rewrite binaries and shared libraries and by describing its application to the binaries used in the CGC. Section V describes existing static binary tools and discusses their similarities to and differences from Zipr.

II. The Zipr Architecture

Figure 1 illustrates the three phases of our binary rewriting method: Intermediate Representation (IR) Construction, Transformation and Reassembly.

The IR Construction phase takes an input program, disassembles it into an IR, deduces indirect branch targets, and prepares the IR for modification. The Transformation phase applies user-specified transforms that programmatically alter the IR to add new functionality to the program. The Reassembly phase converts the modified IR back into executable machine code using a novel technique for program reconstruction.

The IR database (IRDB) mediates communication among cooperating subcomponents. Depending on the task, subcomponents may read, write, or read and write the IRDB. The IRDB is an SQL-based system that stores a variety of information about the binary obtained from different sources. Besides information traditionally represented by an IR (e.g., the control flow graph), the IRDB is designed to store information about the original and modified programs in a way that supports analysis and transformation. Section II-A describes how our approach populates the IRDB with information about the original program and explains the specific features of the IRDB used by Zipr.

A. IR Construction

The IR Construction phase consists of stages normally associated with "binary code analysis" [3]: disassembly and control flow graph (CFG) construction. The IR Construction phase begins with disassembly of the original program's instructions. Next, it reconstructs the original program's CFG and collects associated metadata. The results of both stages are stored in the IRDB for processing by the Transformation and Reassembly phases.

1) Disassembly: Disassembly of a binary program, "[d]ecoding bytes into machine instructions" [3], is not an easy task. A program may have overlapping instructions or non-code bytes may be interspersed among instructions [3]. Disassembly is even more difficult when the binary program does not contain symbols [3].

Our rewriting algorithm does not require perfect disassembly but does rely on accurate disambiguation between code and data. In jump tables or in code that computes on emdedded, read-only data elements, Instructions and data are often mixed [3]. The problem is particularly difficult because bytes of data often decode into valid instructions [3].

In the general case, given access only to a program's machine code, code/data disambiguation is not just difficult, it is impossible [4]. In practice, disambiguation between code and data is feasible but not easy. And although there are many existing disassemblers that disambiguate code and data, none is perfect. Schwartz, et al. give a good overview of two popular types of disassemblers and the techniques each uses to disambiguate code and data [5]. This work by Schwarz et al. also exemplifies the research being done to improve disambiguation techniques, in particular, and disassembly techniques, in general [5].

By virtue of our rewriting technique's modular design, our methodology can aggregate the output of multiple disassemblers. By aggregating the output of the disassemblers, our approach leverages each tool's strengths and compensate for weaknesses. Moreover, the modular design gives our methodology the flexibility to include the output of the new disassemblers built by researchers like Schwarz et al. As of this writing, Zipr combines the output of `objdump` and IDA Pro.

There are four possible outcomes when disambiguating a range of bytes. The disassembler can: 1) correctly and conclusively label the range of bytes as data or instructions; 2) incorrectly but conclusively label the range of bytes as data; 3) ambiguously label the range of bytes or 4) incorrectly but conclusively label the range of bytes as instructions.

Case 1 obviously requires no special handling.

Our rewriting technique handles Cases 2 and 3 conservatively. For Case 2, the range of bytes is labeled as data but they are actually instructions. Our approach handles this case by treating the bytes as both data *and* instructions. We fix the bytes to their original address and use the decoded bytes as instructions for the purpose of CFG construction (Section II-A2).

Case 3 is a byproduct of our rewriting technique's use of multiple disassemblers, as mentioned earlier. When a conclusive determination cannot be made about whether a range of bytes is code or data (i.e., the disassemblers disagree), our approach treats the bytes as both code and data and handles it as an instance of Case 2.

If Case 4 occurs, it is possible that our rewriting technique will generate a non-functioning transformed program. Therefore, it is critical that we be conservative in our analysis of disassemblers' outputs. If there is any chance that a range of bytes labeled instructions actually contains data, we treat the disassemblers' output as if it were inconclusive (Case 3). Even though our approach treats this case conservatively and, in most cases, correctly, Zipr also emits a warning to the user to make debugging easier in the case of failure.

Given the aforementioned caveats, our approach is able to rewrite programs in the most common cases. In support of rewriting, our methodology captures the following features of an instruction during the disassembly stage: 1) fallthrough instruction; 2) target, if it is a control flow instruction and the target is known statically; and 3) pinned address. The fallthrough instruction is used for reassembling linear sequences of instructions and is null if the instruction is an unconditional program control instruction. The target is used for correctly matching instructions that refer to one another by their addresses (See Section II-B1) and is only used if the instruction is a program control instruction whose target is static. An instruction's fallthrough and target instructions are stored in logical format which means that the relationships are independent of the original program's layout. Assignment of the pinned address is discussed next.

2) CFG Construction: Under the conservative assumption of perfect disassembly of a program's binary code, constructing a program's CFG from its machine code is still not a straightforward process. Meng et al. present several cases where CFG construction is complicated even when accurate disassembly is possible: indirect control flow, non-return functions, functions that share code, non-contiguous functions and functions with tail calls [3]. While each complicating factor must be managed and even the best of existing tools (e.g., IDA Pro) are not able to handle every case correctly [3], our technique assumes proper analysis of indirect control flow. Therefore we focus our attention on this complicating factor.

Our technique uses *pinned addresses*, locations of instructions in the original program/library that may be targeted indirectly at runtime. Addresses of units of data are always pinned. On the other hand, the address, a, of an instruction, i, in the original program is pinned if the original program calculates dynamic program control references to a at runtime. In this case, a will be stored as the pinned address value of i. In other words, pinned address analysis depends directly on correct calculation of indirect control flow.

Addresses of instructions may be pinned for a number of reasons. Most commonly, however, addresses are pinned because they are the targets of indirect branches (IB). IB targets (IBTs) appear in jump tables, immediately after call instructions, the beginning of functions, etc. Just because program control

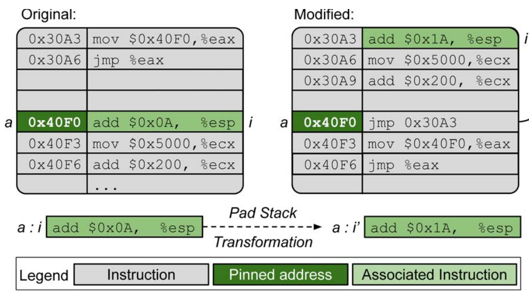

Fig. 2. Pinned addresses, transformations and references.

reaches an instruction indirectly does not mean that its address must be pinned. There are cases where the program's behavior with respect to an IBT can be analyzed and modeled statically.

For our rewriting methodology to operate correctly it is *not* necessary to determine the set of possible targets for every particular indirect branch instruction. Our technique relies only on the fact that P, the set of all pinned addresses, contains *at least* all the addresses of IBTs in the original program. In other words, we rely on the creation of P such that $B \subseteq P$ where B is the set containing the addresses of every IBT from the original program.

It is possible to calculate P naïvely by adding every instruction of the original program to P . This assignment clearly satisfies the requirement. As discussed in Section II-C, such an assignment does not give the reassembly technique the flexibility to re-place instructions. Moreover, it does not allow for the creation of an *efficient* rewritten binary program.

Ideally $B = P$. As $|P - B|$ grows, our method generates an increasingly less space-efficient rewritten binary. Therefore, our algorithm leverages a set of heuristics that analyze the original program's CFG to select pinned addresses. Again, it is imperative that our technique be conservative; missing a pinned address will cause our rewriting algorithm to generate a transformed binary that does not operate correctly.

For a more detailed description of the algorithms used to identify pinned addresses of instructions and data, see Hiser et al. [6] and Zhang et al. [7]. Zipr, is able to handle very complex programs and libraries, even if they include large amounts of handwritten assembly code (see Section IV).

Pinned addresses of instructions play an important role in reassembly. Throughout the rewriting algorithm, a pinned address, a, of an instruction in the original program corresponds to exactly one instruction, i. IR Construction assigns the original correspondence between a and i. During the Transformation phase, one or more transformations may change i to i' and a will correspond to i'. For the modified program to function according to the semantics of the original program, as subsequently modified through user-specified transformations, when the transformed program's program counter (PC) reaches address a, instruction i' must be executed. Section II-C discusses the Reassembly phase which maintains this condition.

Figure 2 shows an example of this process. Instruction i is associated with pinned address a. The illustrative *Pad Stack* transformation modifies i so it allocates a larger stack. The modified instruction, i', is still associated with a. When i' is eventually placed at address `0x30A3` in the modified program, the reference at a is updated appropriately.

B. Transformation

The Transformation phase modifies the original program's IR. Mandatory transforms make it possible for user-specified transforms to modify the original program's IR without regard for the details of the specific target platform. User-specified transforms are optional transformations that modify or add/remove functionality to/from the original program.

1) Mandatory Transformations: Mandatory transformations in the Transformation phase produce a modified IR that enables the reassembly algorithm to place recreated instructions arbitrarily in the modified program's address space.

Mandatory transformations most commonly address issues with the target platform and its ISA. For example, many x86 instructions can use PC-relative addressing. The jump instruction is one such instruction.

Assume instruction i_1 transfers control to i_2 with a jump. On an x86, i_1 is a jump to a_{i_2}, the address of i_2. However, a_{i_2} is encoded in i_1 relative to a_{i_1}. To relocate instructions, relationships like these that rely on the instructions' addresses in the original program have to be translated into logical links. Fortunately, the IR maintains such logical connections among instructions. Returning to the example, the IR links i_1 to i_2, not a_{i_2}. Memory operations may also be PC-relative.

Zipr includes all the required mandatory transformations for the x86-32 and x86-64 platforms.

2) User-specified Transformations: Once the mandatory transformations are applied, user-specified transformations are applied. These are transformations that the user implements that will modify the original program. As mentioned earlier, there are many ways a user could modify the original program to improve its security, reliability and dependability.

Instead of forcing the user to choose from a set of predefined transformations, Zipr provides the user an API to develop their own. The API allows the user to iterate through the functions and instructions of the original program. Users can change (modify or replace) or remove instructions. They can even add new instructions or specify how to link in pre-compiled program code and execute functions therein.

C. Reassembly

At the heart of our approach's novel reassembly technique is an algorithm that carefully reassembles the modified IR into a series of instructions and units of data which are then assigned a location in the modified program's address space.

The process begins by creating references in the modified program at the pinned addresses from the original program. These references target a pinned address' associated instruction or unit of data, as explained in Section II-A. The targets of those references (and their fallthrough instructions) are placed arbitrarily in the remaining free space and the references are marked as resolved. In the process of resolving the initial set of references, new unresolved references may be introduced. The targets of those references are again placed arbitrarily in the remaining free space and the references are resolved. The process continues until there are no more unresolved references.

Figure 3 illustrates the internal state of the algorithm as it reassembles a program. The following subsections explain the reassembly algorithm in detail.

1) Initial Reference Placement: At the outset, the modified program's text segment is empty. To ensure that there is always enough space for additional code that a user may have added in a user-specified transformation, the modified program's address space is augmented with an "infinite" overflow area whose contents will be appended to the rewritten binary as necessary. The data segment is copied directly from the original program. The reassembly algorithm begins by attempting to place unresolved constrained references at pinned addresses.

A *reference* is a link to data or instructions in a *dollop*. A dollop is a linear sequence of instructions linked by their fallthroughs. A reference to a dollop requires that the reassembly algorithm place a control transfer to jump to the referred-to instruction. References are *unresolved* when they link to data or dollops in IR form (*i.e.*, they have not yet been concretized in the final output program.) When a dollop is reconstructed from its IR into instructions and assigned a location in the modified program's address space, references are *resolved* to those particular addresses. In Figure 3, r_1, r_2 and r_3 are references. r_1 and r_2 are resolved and r_3 is unresolved.

A reference to a dollop is *constrained* when there is a restriction on where its target may be placed within the modified program's address space. For example, on machines with PC-relative branches with limited branch offsets, the control transfer for the target dollop is constrained by the size of the branch offset field in the encoded instruction.. Such constraints can be resolved via branch chaining. See Section II-C3 for more details on how branch chaining is used.

In Figure 3, there is a 2-byte instruction i_1 at 0x4000EE and a 3-byte instruction i_2 at 0x4000F0 and both have pinned addresses. The control transfer to the transformed instruction i'_1 will have to encode the address of i'_1 when it is placed. No matter how the ISA encodes addresses (relative or absolute), the encoding cannot exceed two bytes without interfering with the reference at the adjacent pinned address. If the ISA does not support addressing the full address space in two bytes, the reference at 0x4000EE must be constrained.

2) Handling Dense References: On machines with variable length instructions such as x86, it is possible for references in the code to be too close together for a control transfer instruction to fit. For example, on x86 there might be a reference at consecutive addresses, but there is no useful one-byte control transfer instruction.[1]

In such a situation, to resolve the references, our technique uses *sleds*. A sled is a sequence of bytes, any of which can be jumped to, and control of execution re-synchronizes at a known point. For example, on x86, consider the bytes: `0x68, 0x90, 0x90, 0x90, 0x90, 0xf4`. If control is transferred to the first byte of the sequence, the machine executes a `push 0x90909090` instruction (since `0x68` is the push immediate opcode), then the instruction for the `0xf4` byte. If control is transferred to any of the `0x90` bytes, the machine executes 1 or more `nop` instructions (since `0x90` is the opcode for `nop`), then the instruction for the `0xf4` byte. If we need a longer sled to deal with more densely packed references, we can simply add more `0x68` bytes at the beginning of the sequence. The four `0x90`s indicate the end of the sled.

[1]The `ret` instruction is a 1-byte control transfer, but is not suitable for resolving a reference.

Our sleds have another nice property: by examining the value(s) pushed on the stack, we can determine which path was taken through the sled. Carefully crafted dispatch code inspects the stack contents and resolves the references appropriately.

This sled technique can be particularly slow to execute, but are very rarely necessary. Thus, the performance impact is negligible. In practice, we have only observed dense references areas of size 2 or 3. Simple optimizations (such as replacing the nops with a 2-byte control transfer) eliminate any practical need for more sophisticated techniques.

We do note that sleds can likely be used on any architecture with variable-length instructions, However, the sled's exact implementation is necessarily ISA dependent.

3) Reference Expansion and Chaining: Once a constrained unresolved dollop reference is placed at each pinned address, the reassembly algorithm determines which references can be unconstrained. Depending upon the ISA of the implementation target, there is a minimum size, s, necessary to store an instruction that addresses the entire address space. If the space between adjacent constrained unresolved references r_1 and r_2 is greater than s, our algorithm converts r_1 to an unconstrained unresolved reference. In Figure 3, r_2 would initially have been a constrained unresolved reference but has been converted to an unconstrained unresolved reference because there are no pinned addresses in $[\mathtt{0x4000F0}, \mathtt{0x4000F0} + s)$.

For every remaining constrained unresolved reference, r, that references instruction i, a new unresolved unconstrained dollop reference, r', is added in the modified program's address space. r is resolved to r' through one or more intermediate references and r' is set to reference i. This process is known as *chaining* [8]. In Figure 3, r_1 is a reference to d_1 that is chained through r_3.

4) Reference Resolution and Instruction Placement: At this stage of the reassembly, all unresolved references are unconstrained. In addition to the information in the IRDB, the reassembly algorithm relies on three data structures. 1) uDR: The list of unresolved references, 2) D: A list of unplaced dollops (initially empty), and 3) M: A mapping between instructions and their location in the modified program's address space. The final stage of the reassembly algorithm is iterative: Every unresolved reference, r_u, to instruction i in list uDR is considered in turn until the list is empty.

Reference r_u is handled in one of two ways. Either i is already placed in the modified program's address space or it is not. In the former case, the reassembly algorithm simply emits a resolved unconstrained reference to $M[i]$. r_u is removed from uDR and the loop continues. The latter case is more involved — a dollop containing i must be retrieved or constructed and then placed. The reassembly algorithm searches D for d, the dollop containing i.

If no dollop is found, the reassembly algorithm constructs a dollop that contains i. The dollop construction process is straightforward. Dollop d begins with instruction i_0 and

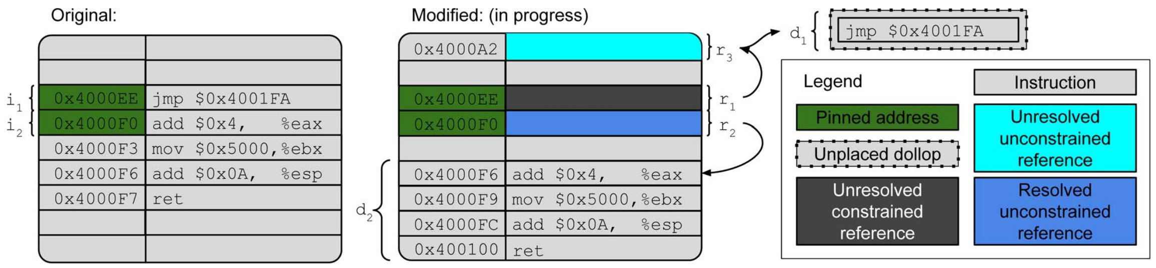

Fig. 3. The reassembly algorithm in the process of reassembling a modified program. Section II-C explains the process and describes this example.

includes i_0's fallthrough i_1, i_1's fallthrough i_2, and so on. The last instruction in d, i_n, is the first instruction that has no fallthrough. The reassembly algorithm places the instructions of d linearly in a consecutive block of addresses. In Figure 3, d_2 is a placed dollop.

When there is no block of free space big enough to accommodate the instructions of d, the dollop may be split. Furthermore, large dollops may be split to fill small blocks of free space. Dollop d of instructions $i_1...i_s...i_n$ is split by choosing a split point, i_s. Dollop d is truncated to contain instructions $\{i_1...i_{s-1}\}$ and d' is built to contain instructions $\{i_s...i_n\}$. An unconstrained unresolved reference r that references i_s is appended to the end of d. The unresolved reference r is added to uDR and d' is added to D.

After d is placed, r_u is resolved and the map M is updated for all instructions in d. Any other unresolved references that target instructions in d are resolved as well. In Figure 3, r_2 is resolved to d_2.

An instruction i_1 in the just-placed dollop d may reference another instruction, i_2. If i_2 is already placed, that reference is resolved immediately. Otherwise, an unresolved reference r_2 is created that references i_2 and r_2 is added to uDR. The modified program is completely reassembled when uDR is empty.

5) Example: Again, Figure 3 illustrates these concepts in the context of a program being reassembled. In this example we will describe Zipr on the x86 ISA where there are variable length instructions. A jump — the implementation of references for this target — can be as short as two bytes (program control transfer is constrained to nearby locations) and as long as five (program control can be transferred anywhere in the program).

In this example there are two pinned addresses, two dollops and three references. Dollop d_2 is already placed; dollop d_1 is not. Reference r_1 began as a constrained unresolved reference to an instruction in dollop d_1. For expository purposes, assume that d_1 could not be placed at an address that is addressable in 2 bytes from r_1. Jump chaining was used and reference r_1 was resolved to r_3 and r_3 became an unresolved unconstrained reference to the instruction in d_1.

Because dollop d_2 is already placed, reference r_2 is resolved. Reference r_2 began as a constrained reference but because there were no pinned addresses in [0x4000F0, 0x4000F5), r_2 was converted to an unconstrained reference.

III. Code Placement

The previous section describes an unoptimized algorithm for placing dollops in a rewritten program. An important design target for CGC (and for many embedded or otherwise resource-constrained environments) is minimizing memory overhead metrics such as the maximum resident set size (MaxRSS).

Pinned addresses play a critical role in this regard. By definition, no matter how our technique rewrites a program or moves code around in memory, the program's execution may jump to a pinned address. For example, if/when it does jump to a pinned address on page a, the operating system pages in all the code on page a, not just the bytes that contain the targeted instruction. To minimize the MaxRSS, the rewriting algorithm should place as much code as possible on pages with pinned addresses and only when those pages are filled should it place code on pages without pinned addresses or in the overflow area. If the rewriting algorithm can place all the code on pages that have pinned addresses then, in this general case, the MaxRSS overhead of the rewritten program will be zero.

Recall the discussion of constrained and unconstrained references from Section II-C. By default, our technique attempts to make all references unconstrained. This provides the maximum amount of flexibility for placing dollops and naturally presents a way of realizing code layout diversity. However, this is not always optimal. If a significant percentage of references point to nearby dollops, using an unconstrained reference wastes space in the output program. The exact amount of space wasted by unconstraining every reference depends on the way that the target ISA encodes those instructions.

Based on LLVM's technique for selecting appropriately sized `jmp` instructions when compiling code for the x86 [9], we wrote a layout algorithm that unconstrains references in a rewritten program only when necessary, thereby favoring memory overhead reduction over layout diversity. For the x86 target, references are expanded, or *relaxed*, from 2-byte `jmp` instructions to 5-byte `jmp` instructions only when the target dollop is greater than 127 bytes away, in either direction. In addition, we place dollops as close to their referents as possible.

Changing the layout algorithms used when rewriting a program does not require the user to recompile or otherwise modify Zipr itself – these layout algorithms are implemented as plugins through Zipr's programming API and SDK.

IV. Evaluation

A. Robustness

To demonstrate the robustness of Zipr, we transformed a large number of x86-64 binary programs and confirmed their functionality. These tests included large code bases such as *libc*, OpenJDK's *libjvm*, the Apache web server, and the Linux core utilities. All timing information reported in this subsection were gathered from experiments run on an Intel Core i7 CPU operating at 3.40GHz with four cores and 8 threads with 8GB of RAM. We applied a *Null Transformation* to these programs and libraries. The Null Transformation is the most basic transformation. It is simply a no-op modification invoked on the IR during the user-specified transformation stage of the Transformation Phase. In other words, the original and the modified programs are semantically equivalent. Consequently, any change to program behavior after it has been rewritten is the result of our rewriting technique.

We emphasize the importance of the Null Transform because the overhead incurred must be incurred by all security transforms and is thus the minimal overhead that a transform might induce.

a) libc: Transforming *libc* is a particularly difficult task. It is a large system library (it contains over 10,000 C source and header files) and the object code is not entirely generated by a compiler. The library source code contains over 38,000 lines of handwritten assembly code – almost 22% of the entire codebase.

As mentioned previously (see Section II-A1), binary program analysis is easier when that program is generated by a compiler because the analysis can take advantage of idioms used by the compiler to generate code. That type of pattern analysis is not possible when disassembling handwritten code.

To test the robustness of our rewriting technique as implemented in Zipr, we used the *libc* unit test system to validate a rewritten version of the library. The unit test system is thorough – it has more than 2500 individual tests. If a rewritten version of the library is able to pass those tests, we can be reasonably sure that the two versions are semantically equivalent (which is the expected behavior because we rewrote with the Null Transformation).

For a baseline, we compiled and built *libc* from source and ran their suite of unit tests. We collected a list of the tests that passed, failed and did not execute. Next, we rewrote *libc* with the Null Transformation using Zipr. *libc* is a 1.6MB library and it took Zipr under 6 minutes to transform. After transformation, we reran the entire suite of unit tests and compared the results. An execution of the unit tests on the native *libc* and the Zipr-rewritten *libc* generated the same results.

b) libjvm: As another demonstration of the robustness of our prototype rewriter, we processed the *libjvm* shared object from Open JDK. This shared object is the core of the Java Virtual Machine on our test system and is over 12MB in size – almost 5 times bigger than *libc*. It took Zipr less than 58 minutes to process a program of this size. Like *libc*, *libjvm* contains handwritten assembly code.

To test the rewritten *libjvm*, we installed *jedit*. Jedit is a full-featured, Java-based text editor. We ran *jedit* under the rewritten *libjvm* and manually performed a variety of common actions, such as opening a file, saving a file, search/replace, inserting new text, deleting text, spell check, etc.

Similar to libc, we also used the appropriate unit tests for libjvm. During all testing we noticed no abnormal slowness in the application, and no faults or errors were observed.

c) Apache: Transforming Apache proved to be another good indicator of the robustness of our rewriting technique. Apache is a large C program containing approximately 56,000 lines of code spread across several shared libraries. For our tests we compiled it using the default compilation flags which include full compiler optimizations and no debugging information. Further, we did tests with and without using position independent code (PIC) in the main executable (shared libraries require PIC). Under this compilation configuration, the binary is 624K and Zipr takes 1 minute and 11 seconds to transform.

Further, we configured Apache to service requests in several different ways. In one configuration to run service requests in multi-process mode where each request is serviced by separate process. In another configuration multiple processes each had multiple threads.

For each compilation mode and configuration, we used Zipr to rewrite both the main binary and the Apache-specific shared libraries. They were transformed as described previously using the Null Transformation.

Finally, we tested the functionality of the transformed versions of Apache. For each compilation mode and configuration we tested the transformed main executable inter-operating with the transformed shared libraries. The tests executed multiple requests in parallel using the Apache Jmeter testing tool and include testing basic functionality, as well as more advanced functionality such as executing CGI scripts to dynamically generate results. Across all configurations, no failures were observed.

B. Using Zipr to Add Security

As stated previously, DARPA's CGC was an excellent venue for assessing Zipr's utility for efficiently transforming binary programs to add additional security protections.

Each competitor in the CGC built a cyber reasoning system (CRS) whose job was to protect a set of previously unseen statically-linked x86 32-bit binary programs (challenge binaries or CBs), during an autonomous game of capture the flag (CTF). The CRS had access to only the binary CBs – no source code, debugging information or other metadata was available for the CRS during analysis. Moreover, the CBs were designed from scratch for the competition by various CB authors.

A CRS defended a CB by adding additional security or reliability. Such an augmented CB was called a replacement challenge binary (RCB). Alternatively, a CRS could also deploy a network filter. Exploits were divided into two categories: control-flow hijacking and information disclosure attacks. Our team's strategy was to handle the former by rewriting CBs to augment them with a simple form of control-flow integrity

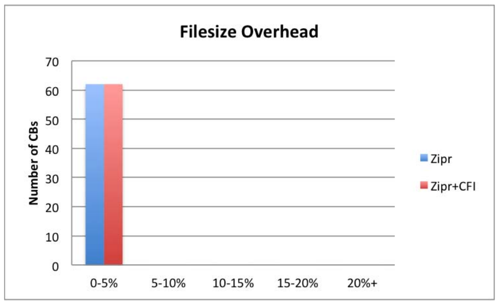

Fig. 4. Histogram of Filesize Overhead.

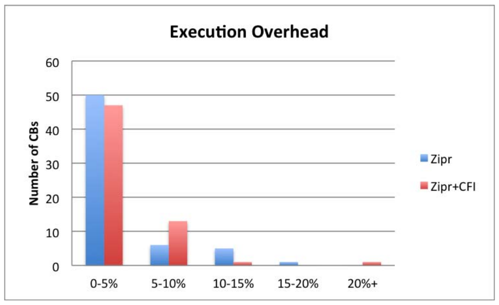

Fig. 5. Histogram of Execution Overhead.

(CFI) [10], and the latter by deploying network filters.[2]

For the CTF game, CRS's were scored on the characteristics of the RCBs. Those scores were compiled and the team with the best score at the end of the game was declared the winner of the final event (CFE). Characteristics of each RCB that affected scoring were calculated relative to the baseline performance of the CB. To deterministically measure the baseline performance, CGC employed a) a specialized operating environment, the DARPA Experimental Cyber Research Evaluation Environment (DECREE), and b) pollers for each CB.

DECREE is a Linux-based operating system with various restrictions (*e.g.*, only seven system calls, limitations on inter-process communication, no filesystem or network access) [2]. A CB poller is an input to a CB. DARPA mandated that CB authors provide a pollers to exercise all the functionality of a CB. In combination, the DECREE environment and the pollers made it possible to generate characteristics (execution time, file size, runtime memory usage, and functionality) of each CB.

DARPA had strict targets for each of these characteristics: 20% overhead for file size, 5% overhead for performance and memory and limited loss of functionality. Any overhead beyond those thresholds negatively impacted a CRS's score. Because Zipr was the base rewriting technology for our defenses, its efficiency was vital to Xandra's performance in CFE. Our second place finish in CFE showed that Zipr performs well under these strict constraints. Besides finishing second overall, Xandra had the best defensive score among all competitors.

For 62 of the CBs deployed during CFE, we calculated the file size, performance and memory overhead. Figures 4, 5, and

[2]Despite known weaknesses with CFI approaches [11], [12], we felt that even a simple form of CFI would be sufficient to thwart attacks as competitors would need to bypass CFI in an automated manner. Our bet paid off, our CRS was only breached once using a control-flow hijacking attack during the entire competition.

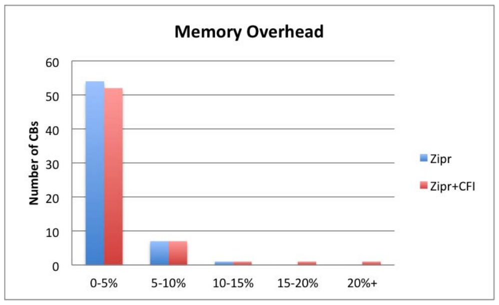

Fig. 6. Histogram of Memory Overhead.

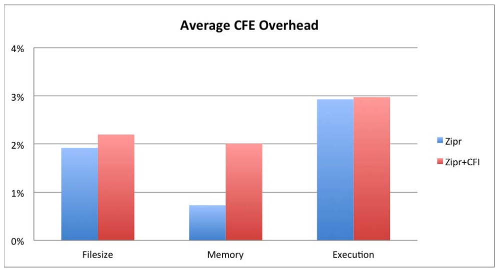

Fig. 7. Average overheads for final event CBs.

6 show a histogram view for each metric for both the baseline Zipr and Zipr augmented with CFI.

With respect to filesize, both configurations incurred overhead of less than 5%, well within the target 20% threshold. With respect to execution overhead, the vast majority of CBs for the baseline Zipr configuration was within 5%, though several CBs incurred overhead between 5% and 20%. The overhead attributable to CFI can be readily seen by a slight reduction in the number of CBs within the 5% threshold and a corresponding increase across all other performance bins. For memory, the majority of CBs for both configurations remained within 5%, though the CFI configuration resulted in additional memory pressure as readily seen in Figure 6. In particular, one CB resulted in more than 50% memory overhead for CFI. This CB elicited a pathological behavior in our code placement algorithm because of the number of pinned addresses and its relatively large dollops. The pinned addresses fragmented the program's address space which made it hard to fit those dollops inside the fragments. As a result, many of the dollops ended up being placed in the overflow area (see Section II-C1).

Finally, Figure 7 shows the average filesize, memory and execution overhead across the CBs. Overall, we were generally able to achieve low overheads across all three metrics, for both the baseline Zipr and Zipr augmented with CFI configurations.

We have successfully used Zipr to perform stack randomization [13], dynamic canary randomization [14], and dynamic code mixing similar to Binary Stirring by Wartell, et al. [15] as well as a variety of whole program randomization. [16] Unfortunately, space prohibits a thorough evaluation of these techniques and others that Zipr can apply to secure COTS.

V. Related Tools

Providing a tool for improving binary programs, especially their security, reliability and dependability, is very important and many techniques have been investigated. We provide a brief discussion of some popular binary rewriting techniques, organized into dynamic and static rewriting tools.

Unlike static rewriters, dynamic binary rewriters, sometimes referred to as software dynamic translation systems, modify binary programs at runtime [17], [18], [19]. Unfortunately, such translation can be costly in terms of memory and execution-time overhead. In addition, the restricted DECREE environment precluded the use of any dynamic rewriting techniques, including recent rewriters that support CFI [20].

Existing static binary rewriters include SecondWrite, Etch, Vulcan, REINS, and PSI [7], [21], [22], [23], [24]. Many static rewriters require additional program information to operate correctly (e.g., source code or object code), which may not be available for legacy software. Most static rewriters keep an original copy of the program's code as a hedge against failure when they cannot correctly disambiguate code from data. This limitation alone would have precluded their use in CGC. Others cause significant performance degradation. These space and time overheads can make a rewriter unsuitable for servers and embedded systems. Zipr and improves on the state of the art by requiring no development artifacts and having low size and performance overheads.

VI. Conclusion

This paper presented an efficient method for statically modifying programs and libraries using only their binary code. We outlined the system's algorithm for binary rewriting and detailed its novel technique for reassembly. The evaluation of Zipr demonstrated that our methodology does indeed generate efficient modified programs. The results from the application of Zipr in Xandra during CGC further demonstrate the technique's ability to generate efficient modified programs.

Acknowledgment

This work is based on research sponsored by DARPA under agreement numbers FA8750-15-C-0118 and FA8750-14-C-0110 and AFRL under agreement number FA8750-15-2-0054.

References

[1] T. J. Edler von Koch, B. Franke, P. Bhandarkar, and A. Dasgupta, "Exploiting function similarity for code size reduction," *SIGPLAN Not.*, vol. 49, no. 5, pp. 85–94, Jun. 2014.

[2] M. Walker, "Machine vs. Machine: Lessons from the First Year of Cyber Grand Challenge — USENIX," 2015. [Online]. Available: https://www.usenix.org/node/190798

[3] X. Meng and B. P. Miller, "Binary code is not easy," University of Wisconsin, Tech. Rep. [Online]. Available: ftp://ftp.cs.wisc.edu/paradyn/papers/Meng15Parsing.pdf

[4] M. Prasad and T. Chiueh, "A binary rewriting defense against stack based buffer overflow attacks," in *Proceedings of the General Track: 2003 USENIX Annual Technical Conference*, 2003, pp. 211–224.

[5] B. Schwarz, S. Debray, and G. Andrews, "Disassembly of executable code revisited," in *Proceedings Ninth Working Conference on Reverse Engineering (WCRE) 2002*, 2002, pp. 45–54.

[6] J. D. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson, "ILR: Where'd my gadgets go?" in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, ser. SP '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 571–585.

[7] M. Zhang, R. Qiao, N. Hasabnis, and R. Sekar, "A platform for secure static binary instrumentation," in *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '14. New York, NY, USA: ACM, 2014, pp. 129–140.

[8] B. W. Leverett and T. G. Szymanski, "Chaining span-dependent jump instructions," *ACM Trans. Program. Lang. Syst.*, vol. 2, no. 3, pp. 274–289, Jul. 1980.

[9] E. Bendersky, "Assembler relaxation," 2013. [Online]. Available: http://eli.thegreenplace.net/2013/01/03/assembler-relaxation

[10] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 1, pp. 4:1–4:40, Nov. 2009. [Online]. Available: http://doi.acm.org/10.1145/1609956.1609960

[11] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *Proceedings of the 24th USENIX Conference on Security Symposium*, ser. SEC'15. Berkeley, CA, USA: USENIX Association, 2015, pp. 161–176. [Online]. Available: http://dl.acm.org/citation.cfm?id=2831143.2831154

[12] M. Zhang and R. Sekar, "Control flow integrity for cots binaries," in *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. Washington, D.C.: USENIX, 2013, pp. 337–352.

[13] B. D. Rodes, A. Nguyen-Tuong, J. D. Hiser, J. C. Knight, M. Co, and J. W. Davidson, "Defense against stack-based attacks using speculative stack layout transformation," in *Runtime Verification*, ser. Lecture Notes in Computer Science, S. Qadeer and S. Tasiran, Eds. Springer Berlin Heidelberg, 2013, vol. 7687, pp. 308–313.

[14] W. H. Hawkins, J. D. Hiser, and J. W. Davidson, "Dynamic canary randomization for improved software security," in *Proceedings of the 11th Annual Cyber and Information Security Research Conference*. ACM, 2016, p. 9.

[15] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 157–168.

[16] M. Co, J. W. Davidson, J. D. Hiser, J. C. Knight, A. Nguyen-Tuong, W. Weimer, J. Burket, G. L. Frazier, T. M. Frazier, B. Dutertre *et al.*, "Double Helix and RAVEN: A system for cyber fault tolerance and recovery," in *Proceedings of the 11th Annual Cyber and Information Security Research Conference*. ACM, 2016, p. 17.

[17] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa, "Retargetable and reconfigurable software dynamic translation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 36–47.

[18] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. 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*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 190–200.

[19] D. Bruening, T. Garnett, and S. Amarasinghe, "An infrastructure for adaptive dynamic optimization," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 265–275.

[20] M. Payer, A. Barresi, and T. R. Gross, "Lockdown: Dynamic control-flow integrity," *CoRR*, vol. abs/1407.0549, 2014. [Online]. Available: http://arxiv.org/abs/1407.0549

[21] K. Anand, M. Smithson, A. Kotha, K. Elwazeer, and R. Barua, "Decompilation to compiler high IR in a binary rewriter," University of Maryland, Tech. Rep., November 2010. [Online]. Available: http://www.ece.umd.edu/~barua/high-IR-technical-report10.pdf

[22] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Bershad, and B. Chen, "Instrumentation and optimization of Win32/Intel executables using etch," in *Proceedings of the USENIX Windows NT Workshop*, ser. NT'97. Seattle, Washington: USENIX Association, August 1997.

[23] A. Srivastava, A. Edwards, and H. Vo, "Vulcan: Binary transformation in a distributed environment," Microsoft Research, Tech. Rep. MSR-TR-2001-50, April 2001.

[24] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Securing untrusted code via compiler-agnostic binary rewriting," in *Proceedings of the 28th Annual Computer Security Applications Conference*, ser. ACSAC '12. New York, NY, USA: ACM, 2012, pp. 299–308.