Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics

Erick Bauman
University of Texas at Dallas
erick.bauman@utdallas.edu

Zhiqiang Lin
University of Texas at Dallas
zhiqiang.lin@utdallas.edu

Kevin W. Hamlen
University of Texas at Dallas
hamlen@utdallas.edu

***Abstract*—Static binary rewriting is a core technology for many systems and security applications, including profiling, optimization, and software fault isolation. While many static binary rewriters have been developed over the past few decades, most make various assumptions about the binary, such as requiring correct disassembly, cooperation from compilers, or access to debugging symbols or relocation entries. This paper presents MULTIVERSE, a new binary rewriter that is able to rewrite Intel CISC binaries without these assumptions. Two fundamental techniques are developed to achieve this: (1) a superset disassembly that completely disassembles the binary code into a superset of instructions in which all legal instructions fall, and (2) an instruction rewriter that is able to relocate all instructions to any other location by mediating all indirect control flow transfers and redirecting them to the correct new addresses. A prototype implementation of MULTIVERSE and evaluation on SPECint 2006 benchmarks shows that MULTIVERSE is able to rewrite all of the testing binaries with a reasonable runtime overhead for the new rewritten binaries. Simple static instrumentation using MULTIVERSE and its comparison with dynamic instrumentation shows that the approach achieves better average performance. Finally, the security applications of MULTIVERSE are exhibited by using it to implement a shadow stack.**

I. INTRODUCTION

In many systems and security applications, there is a need to statically transform COTS binaries. Software fault isolation (SFI) [41], including Control Flow Integrity (CFI) [4], constrains the program execution to only legal code by rewriting both data accesses and control flow transfer (CFT) instructions. Binary code hardening (e.g., STIR [46]) rewrites and relocates instructions, randomizing their addresses to mitigate control flow hijacks. By lifting binary code to an intermediate representation (e.g., LLVM IR), various compiler-missed platform-specific optimizations can also be performed [5].

Given so many applications centered around binary code transformation, significant efforts have been made over the past few decades to develop various binary rewriters, particularly for Intel x86/x64 architectures due to their dominance in modern computing. Early approaches for transforming these binaries require special support from compilers or make compiler-specific assumptions. For instance, SASI [17] and PITTSFIELD [27] only recognize `gcc`-produced assembly code—not in-lined assembly from `gcc`. CFI [4] and XFI [18] rely upon compiler-supplied debugging symbols to rewrite binaries. Google's Native Client (NACL) [49] requires a special compiler to compile the target program, and also limit API usage to NACL's trusted libraries. These restrictions have blocked binary rewriting from being applied to the vast majority of COTS binaries or to more general software products.

More recent approaches have relaxed the assumption of compiler cooperation. STIR [46] and REINS [47] rewrite binaries using a reassembling approach without compiler support; however, they still rely upon imperfect disassembly heuristics to handle several practical challenges, especially for position-independent code (PIC) and callbacks. CCFIR [51] transforms binaries using relocation metadata, which is available in many Windows binaries. SECONDWRITE [30] rewrites binaries without debugging symbols or relocation metadata by lifting the binary code into LLVM bytecode and then performing the rewriting at that level. However, it still assumes knowledge of well-known APIs to handle callbacks, and uses heuristics to handle PIC. Lifting to LLVM bytecode can also yield large overheads for binaries not easily representable in that form, such as complex binaries generated by dissimilar compilers. BINCFI [53] presents a set of analyses to compute the possible indirect control flow (ICF) targets and limit ICF transfers to only legal targets. However, BINCFI can still fail when code and data are intermixed. Recently, UROBOROS [43] presented a set of heuristics to recognize static memory addresses and relocate and reassemble them for binary code reuse, but experimental results still show it has false positives on the SPEC2006 benchmarks.

Thus, nearly all static Intel CISC binary rewriters in the literature to date rely upon various strong assumptions about target binaries in order to successfully transform them. While each is suitable for particular applications, they each lack generality. End users cannot be confident of the correctness of the rewritten code, since many of the algorithms' underlying assumptions can be violated in real-world binaries. To advance the state-of-the-art, we present MULTIVERSE, an open source, next generation binary rewriter that is able to statically rewrite x86 binaries without heuristics; binaries rewritten without heuristics have the same semantics as the original.

To this end, we address two fundamental challenges in COTS binary rewriting: (1) how to disassemble the binary code

This work is supported in part by AFOSR awards FA9550-14-1-0119 and FA9550-14-1-0173, NSF awards 1453011 and 1513704, and ONR awards N00014-14-1-0030 and N00014-17-1-2995.

Network and Distributed Systems Security (NDSS) Symposium 2018
18-21 February 2018, San Diego, CA, USA
ISBN 1-1891562-49-5
http://dx.doi.org/10.14722/ndss.2018.23300
www.ndss-symposium.org

and cover all legal instructions, and (2) how to reassemble the rewritten instructions and preserve the original program semantics. To solve the first challenge, we propose a superset disassembling technique, through which each offset of the binary code is disassembled. Such disassembling creates a (usually strict) superset of all reachable instructions in the binary. The intended reachable instructions are guaranteed to be within the superset, thereby achieving complete recovery of the legal intended instructions (i.e., completeness).

To address the second challenge, we borrow an instruction reassembling technique from dynamic binary instrumentation (DBI) [26], which mediates all the *indirect CFT* (iCFT) instructions and redirects their target addresses to the rewritten new addresses by consulting a mapping table from old addresses to new rewritten addresses created during the rewriting. Since all iCFTs are instrumented in a very similar way to how dynamic binary instrumentation rewrites the binary at runtime, the original program semantics are all preserved, achieving soundness (i.e., all program semantics, including control flow destinations, are identical to the original binary). Therefore, our approach is sound and complete with respect to the original static binary's intended instructions and execution semantics.

In summary, our main contributions are as follows:

- We present MULTIVERSE, the first static binary rewriter built on a foundation of both soundness and completeness, raising assurance in the correct execution of rewritten binaries.
- We design a superset disassembling technique, which does not make any assumptions on where a legal instruction should start and instead disassembles and reassembles each offset, achieving complete recovery of original instructions.
- We also develop a static instruction reassembling technique, which translates all indirect control flow transfer instructions (including those in the library) and redirects their target addresses to correct ones, achieving the soundness of original program execution.
- We have implemented these techniques in our prototype, and evaluated it with the SPECint 2006 benchmark suite. Experimental results show that MULTIVERSE correctly rewrites all the test binaries. A comparison with dynamic instrumentation also shows that the static instrumentation enabled by MULTIVERSE has better average performance.
- We have also demonstrated one security application of using MULTIVERSE to implement a shadow stack. In doing so we provide a sample of the possibilities of the security applications of MULTIVERSE.

II. BACKGROUND AND OVERVIEW

A. Scope and Assumptions

The goal of this paper is to develop a new binary transformation algorithm that improves the practicality and generality of existing code transformation applications, such as binary code hardening. Our approach generalizes to arbitrary OSes and Intel-based CISC architectures, but for expository simplicity we here focus on 32-bit x86 binaries running atop Linux (ELF-32) generated by mainstream compilers such as `gcc` or `llvm`. We assume no restriction of original program source code, which can even be hand-written assembly. Although we focus on mainstream compilers for our presentation, our approach accommodates most statically obfuscated binaries (e.g., instruction aliasing, code and data interleaving, etc.). We do not automatically support code that loads shared libraries dynamically, such as with `dlopen`. However, such binaries can still be rewritten after manual recovery of dynamically loaded libraries. In addition, like all existing static binary rewriters, we do not handle any self-modifying or packed code (such as self-extracting compressed software) or JIT-compiled code. Support for such code requires dynamic rewriting since such code is not visible or does not exist in a static binary.

We focus on x86 instead of x64 because legacy x86 applications are less likely to have source available, and many code transformations target older legacy code. In addition, while the differences between x86 and x64 for the purposes of binary rewriting are not too significant, there are some engineering differences that distract from the discussion of binary rewriting. Therefore, we focus on x86 for the purposes of this paper.

B. Challenges

There are enormous challenges in designing a general binary rewriter. To illustrate these challenges clearly, Figure 1 presents a contrived working example. This simple program sorts an array of strings in ascending or descending order (depending on the least significant bit of the program's `pid`) using libc's `qsort` API. When printing out the mode (ascending or descending sort) or printing each array element, the program uses the function `get_fstring`, defined in `fstring.asm`, to determine the format string it should use. This function is written in assembly to show a simple example of interleaved code and data. With this working example, we can organize the challenges into the following categories:

C1: Recognizing and relocating static memory addresses. Compiled binary code often refers to fixed addresses, especially for global variables. Code transformations that move these targets must update any references to them. However, it is very challenging to recognize these address constants within disassembled code and data sections, since there is no syntactic distinction between an address and an arbitrary integer value.

In our working example, the `modes` variable (line 17 of `sort.c`) is an array of function pointers stored in the `.data` section at address `0x0804a03c`. Were we to move `.data`, we would need to identify and change all references to this array to point to its new location. In more complicated applications, it is difficult to reliably differentiate between a pointer-like integer and a pointer—a major challenge in static binary rewriting.

C2: Handling dynamically computed memory addresses. In addition to static memory addresses, there are also dynamically computed memory addresses. A particular challenge concerns iCFTs whose target addresses are computed at runtime. For instance, an indirect jump target can be computed from a base address plus an offset, and a function pointer can be initialized to a function address also computed at runtime. These pointers can even undergo arbitrary binary arithmetic, be encoded (e.g., using a hash table), or be dereferenced in a number of layers (e.g., double pointers or triple pointers), before they are used. Unlike direct CFTs whose targets are explicit, iCFT targets often cannot be predicted statically. Remapping iCFT targets reliably is therefore a central challenge for binary rewriting.

```
// gcc -m32 -o sort cmp.o fstring.o sort.c
#include <stdio.h>
#include <unistd.h>

extern char *array[6];
int gt(void *, void *);
int lt(void *, void *);
char* get_fstring(int select);

void mode1(void){
    qsort(array, 5, sizeof(char*), gt);                    C4
}
void mode2(void){
    qsort(array, 5, sizeof(char*), lt);                    C4
}
                                                           C1
void (*modes[2])() = {mode1, mode2};

void main(void){
    int p = getpid() & 1;
    printf(get_fstring(0),p);
    (*modes[p])();                                         C2
    print_array();
}
```

(a) Source code of **`sort.c`**

```
 1 ;nasm -f elf fstring.asm
 2 BITS 32
 3 GLOBAL get_fstring
 4 SECTION .text
 5 get_fstring:
 6     mov eax,[esp+4]
 7     cmp eax,0
 8     jz after
 9     mov eax,msg2
10     ret
11 msg1:
12     db 'mode: %d', 10, 0                                C3
13 msg2:
14     db '%s', 10, 0                                      C3
15 after:
16     mov eax,msg1
17     ret
```

(b) Source code of **`fstring.asm`**

```
// gcc -m32 -c -o cmp.o cmp.c -fPIC -O2
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char *array[6] = {"foo", "bar", "quuz", "baz", "flux"};     C1
char* get_fstring(int select);

void print_array(){
    int i;
    for (i = 0; i < 5; i++){
        fprintf(stdout, get_fstring(1), array[i]);          C5
    }
}
int lt(void *a, void *b){
    return strcmp(*(char **) a, *(char **)b);
}

int gt(void *a, void *b){
    return strcmp(*(char **) b, *(char **)a);
}
```

(c) Source code of **`cmp.c`**

```
8048510 <print_array>:
...
8048515:   53                    push   %ebx
8048516:   e8 b1 00 00 00        call   80485cc <__i686.get_pc_thunk.bx>
804851b:   81 c3 d9 1a 00 00     add    $0x1ad9,%ebx               C5
8048521:   83 ec 1c              sub    $0x1c,%esp
8048524:   8b ab fc ff ff ff     mov    -0x4(%ebx),%ebp            C5
...
80485a0 <gt>:
80485a0:   53                    push   %ebx
...
80485cc <__i686.get_pc_thunk.bx>:
80485cc:   8b 1c 24              mov    (%esp),%ebx                C5
80485cf:   c3                    ret

80485d0 <get_fstring>:
80485d0:   8b 44 24 04           mov    0x4(%esp),%eax
80485d4:   83 f8 00              cmp    $0x0,%eax
80485d7:   74 14                 je     80485ed <after>
80485d9:   b8 e9 85 04 08        mov    $0x80485e9,%eax
80485de:   c3                    ret
80485df:   6d                    insl   (%dx),%es:(%edi)           C3
80485e0:   6f                    outsl  %ds:(%esi),(%dx)
80485e1:   64 65 3a 20           fs cmp %fs:%gs:(%eax),%ah
...
80485f4 <mode1>:
...
80485fa:   c7 44 24 0c a0 85 04  movl   $0x80485a0,0xc(%esp)       C4
8048601:   08
8048602:   c7 44 24 08 04 00 00  movl   $0x4,0x8(%esp)
8048609:   00
804860a:   c7 44 24 04 05 00 00  movl   $0x5,0x4(%esp)
8048611:   00
8048612:   c7 04 24 24 a0 04 08  movl   $0x804a024,(%esp)
8048619:   e8 12 fe ff ff        call   8048430 <qsort@plt>
...
804864c <main>:
...
8048678:   e8 73 fd ff ff        call   80483f0 <printf@plt>
804867d:   8b 44 24 1c           mov    0x1c(%esp),%eax
8048681:   8b 04 85 3c a0 04 08  mov    0x804a03c(,%eax,4),%eax
8048688:   ff d0                 call   *%eax                      C2
...
```

(d) Partial binary code of **`sort`**

```
Hex dump of section '.rodata':
  0x08048768 03000000 01000200 666f6f00 62617200 ........foo.bar.
  0x08048778 7175757a 0062617a 00666c75 7800     quuz.baz.flux.
```

(e) Hexdump of **`ro.data`** section

```
Hex dump of section '.data':
  0x0804a01c 00000000 00000000 70870408 74870408 ........p...t...
  0x0804a02c 78870408 7d870408 81870408 00000000 x...}...........
  0x0804a03c f4850408 20860408                   .... ...          C1
```

(f) Hexdump of **`.data`** section

Fig. 1: A contrived working example that covers major challenges in x86 COTS binary rewriting.

When our working example calls one of the function pointers in the `modes` array, it is difficult to reliably predict which function will be called until runtime. As shown in the assembly, the `mov` at `0x804867d` sets `eax` to the value of stack variable `p`, which determines the mode (0 or 1). The `mov` instruction at `0x8048681` then assigns `eax` the address held in the index of array `modes` determined by the mode (the address held at `0x0804a03c+0` or `0x0804a03c+4`). Finally, it calls the address in `eax`. Statically predicting which addresses to update, while possible in this simple example, can quickly become intractable (e.g., if the array were dynamically allocated with unknown length).

C3: Differentiating code from data. In x86, there is no syntactic distinction between code and data within binaries [9]. More specifically, code and data can be interleaved. This is typical in hand-written assembly, and in modern compilers that aggressively interleave static data within code sections for performance reasons. Also, code bytes are unaligned—they can start at any offset within executable segments.

Lines 12 and 14 of `fstring.asm` exhibit data bytes amid code bytes. Linear sweep-based disassemblers often misinterpret these as code bytes, resulting in disassembly errors that yield garbage instructions and omit subsequent reachable instructions (e.g., the last `mov` instruction on line 16). Such garbage instructions can be seen in the disassembly starting at address `0x80485df`. While a disassembler using recursive traversal can follow the control flow from the `jz` instruction to avoid some of these errors, a more complicated program with indirect control flow to the `after` label would make it difficult to statically determine which offsets are valid.

C4: Handling function pointer arguments (e.g., callbacks). Functions that expect function pointers as arguments can fail after binary transformation if the referant code is moved but the referring argument is not updated accordingly. Function pointer arguments are usually used in callbacks, where a code pointer is passed from the program as a computed jump destination. Unlike typical dynamically computed memory addresses (C2), which are visible to the rewritten binary, callback pointers are often used in library spaces. As mentioned in C1, it is already challenging to recognize static memory addresses, and it is even more challenging to recognize arguments with function pointer types at the binary level.

Our working example includes a call to the `libc` function `qsort`, which expects a callback function as its last argument, at lines 11 and 14 of `sort.c`. It uses this function to compare each element pair when it sorts the array, and the user must provide a comparison function that is meaningful for the array argument. In the example, the function supplied depends on the `mode` (ascending/descending). The assembly for the call for `mode` 1 is shown starting at address `0x80485fa`. The `mov` instruction at that address moves the address of the `gt` function (`0x80485a0`) on the stack as the argument for `qsort`. If we move the location of `gt` but do not modify this address, `qsort` will call the wrong code.

C5: Handing PIC. While mainstream compilers generate mainly position dependent code by default, they can also generate PIC, which can be loaded at arbitrary addresses. PIC is typically achieved via instructions that dynamically compute their own addresses and expect to find other instructions or variables at known relative offsets. These instructions can break the program if a rewriter fails to identify them.

We compile `cmp.c` in the working example with the `gcc` flag `-fPIC`, which ensures that all functions in that file are compiled as PIC. The results are shown in the disassembly of our working example. Since PIC uses its own address to compute offsets, it uses a `call` instruction to compute its own position in the form of a special function that retrieves the instruction pointer. This function, `__i686.get_pc_thunk.bx`, is shown at `0x80485cc`, and consists only of an instruction that saves the return address into `ebx`. The `print_array` function uses this address to compute the address of `array`. Relocating this code without any modifications causes an incorrect address to be computed, usually resulting in a crash.

C. Key Insights

Binary rewriting is not a new problem. Over the past few decades, a tremendous amount of effort has been devoted to developing various binary rewriters for different purposes under different constraints. Drawing from these existing efforts, including related works using dynamic binary instrumentation (e.g., the widely used PIN [26]), we have derived and systematized the following key insights to address each of the above challenges.

S1: Keeping original data space intact. We can strategically avoid the need to recognize static memory addresses of data if we retain and preserve all bytes that the program might read as data. Since code sections might contain data bytes, we can preserve such data by retaining an old copy of each code section at its original location. For security-focused applications, we can set the original code section non-executable. This approach is used by several existing rewriters (e.g., SECONDWRITE [30], BINCFI [53], STIR [46], and REINS [47]).

S2: Creating a mapping from the old code space to the new rewritten code space. As discussed in C2, there are various forms of dynamically computed memory addresses. Heuristic approaches that attempt to statically identify base addresses and then update each associated offset accordingly are unreliable, since x86 address spaces are typically flat, allowing any base address to potentially index any higher address. Fortunately, we have another unique key observation: *Instead of identifying the base addresses and rewriting them to point to the new location, we can just focus on the final target addresses and ignore how it is computed.*

More specifically, even though a target address can be encoded or computed through many layers of pointers, its final runtime value must eventually flow to the iCFT as its argument. (Note that direct CFTs are not a problem since their target addresses are explicit.) Therefore, if we can map each possible destination address in the old address space to an address in our new, rewritten code, and if we make the mapping available at runtime, then we can rewrite each iCFT to look up the new address immediately after the old destination address has been computed. This allows us to automatically solve C2 without relying on any heuristics.

S3: Brute force disassembling of all possible code. Disassembling is a perennial problem for static binary analyses. Unlike many prior efforts, we observe that *while it is challenging to correctly disassemble arbitrary code, we can instead find a superset of the disassembled code (by brute force disassembling every executable byte offset), and the result will contain the correct disassembly somewhere in the set* (which is why we call our approach superset disassembly). While disassembling from each offset has been explored in malware analysis [21] [48] [22], the resulting disassemblies are intended for reverse-engineering obfuscated code, finding function entry/exit points, and other analysis purposes; no attempt has been made to link the superset code and make it runnable. A new challenge for us therefore concerns linking the instructions in the superset. Fortunately, if we translate all iCFT instructions (as we do in S2), then we can link them together by using our old address to new address mapping table.

S4: Rewriting all user level code including libraries. One possible solution to C4 is to identify each function that uses function pointer arguments in external libraries and patch the function pointer address so that callbacks correctly reference our new text section. Many prior rewriters use this approach, including STIR [46], REINS [47], and SECONDWRITE [30]. However, if our transformation algorithm is sufficiently general, we can instead expand our rewriting to include all program code including libraries. All callbacks will then be executed correctly (by S2) without having to identify callback arguments.

This solution also has the considerable benefit of accommodating C++ exceptions, wherein the `.eh_frame` holds information about exception handler addresses, which may be called from a different module than the caller, effectively acting like a callback. By rewriting the instruction that jumps to the exception handler, C++ exceptions are transparently handled.

S5: Rewriting all `call` instructions in order to handle PIC. It is challenging to identify PIC in the binary code, because there are a great diversity of instructions that derive code

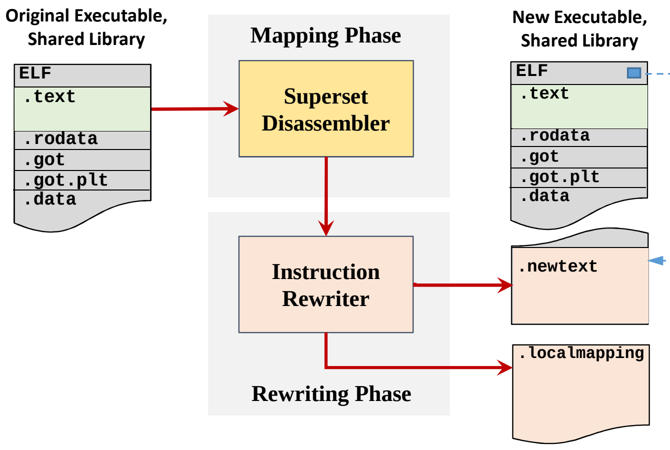

Fig. 2: An Overview of MULTIVERSE.

or data offsets from PIC-computed self-addresses. However, after careful examination of x86 instruction semantics, we find that only the `call` instruction, which pushes the instruction pointer onto the stack, can be feasibly used to compute the base address used in subsequent PIC offset calculations. Therefore, we translate each `call` instruction in the original code into an explicit `push` of the *old (unmodified)* return address followed by a `jmp` to a new, rewritten address (computed from the old target address by querying the mapping table). This transparently preserves PIC because any subsequent address arithmetic will compute a correct old code address, which will be correctly remapped to a correct new address when it finally flows to an iCFT (by S2). If PIC is used to access data, then the correct data is accessed because the pushed address does not flow to an iCFT, and is therefore not remapped.

D. Overview

From the insights above, we have created MULTIVERSE[1], a binary rewriter that accepts an ELF-32 binary or shared library and transforms it to produce a new rewritten binary. As illustrated in Figure 2, our system consists of two separate phases: the *mapping* phase, and the *rewriting* phase.

In the *mapping* phase, we use our *superset disassembler* to disassemble the binary at every byte offset starting from the lowest code address. This produces many copies of the same code, since instruction sequences at most offsets eventually align with an instruction disassembled from a previous offset. We avoid this unnecessary duplication by ceasing disassembly at the first redundant offset of each sequence, and later inserting an unconditional jump to the code previously disassembled for that offset. At a high level, we generate our mapping by disassembling each instruction, determining the length the rewritten instruction will have in the final code, and then using this information to create a final mapping from old addresses to new addresses that will be used in the next phase and placed into `.localmapping`.

In the *rewriting* phase, our *instruction rewriter* again iterates through each disassembled instruction and generates the new bytes that will be placed in the `.newtext` section. We must make this second pass because we cannot generate the final code without already knowing the complete mapping. For instance, if an instruction in the old code refers to a specific offset at a higher address that we have not yet disassembled, we will not know what new offset to use when rewriting the instruction.

Once the new text section is created, we pass it to our ELF writer (which is not shown in Figure 2), which takes the new entry address, mapping, and rewritten text and creates the final rewritten binary. The ELF writer modifies the ELF header and PHDRs in order to create a new segment to hold the new text. The original `.text` section is left in its original location as non-executable data to support reads from the `.text` section (e.g., for jump tables). Next, we present the detailed design of MULTIVERSE. We first describe our *mapping* in §III, and then explain our *rewriting* in §IV.

III. MAPPING

A. Superset Disassembler

"When in doubt, use brute force." – Ken Thompson

Our *superset disassembler* disassembles instruction sequences starting from every byte offset in the binary's text section. This approach can also be considered a form of brute-force disassembly, i.e., we are finding the intended sequences of instructions by brute-force disassembling every possible offset. However, without any further refinement, this would produce a huge number of duplicate subsequences. Therefore, we keep a list of all offsets we have already disassembled, and we stop disassembly from an offset if we reach an instruction we have already encountered. We can do this because we can connect a sequence of instructions to another in our rewriting phase via insertion of an unconditional jump.

A high-level illustration of how the brute-force disassembly process works is shown in Figure 3 and algorithm 1. We start disassembling from offset zero and disassemble instructions until we reach an illegal instruction. Although we could stop disassembling at `jmp` or `ret` instructions (and disassemble the bytes after that instruction in a later pass), we simply try to disassemble as many as possible in each pass, partly for simplicity and partly for code locality reasons. By disassembling as many in a sequence as possible and not breaking up our rewritten code into distant chunks, we are able to benefit from locality in some cases when a program is using short unconditional jumps without having to do advanced analyses. That said, while we currently try to keep long contiguous instruction sequences, we have no restriction on how we organize the new instructions. For example, we could easily break longer sequences into blocks of any size by inserting `jmp` instructions in our rewritten code. Since every instruction in the old code is mapped, we can then move each block to any location in the new code space. This gives us flexibility depending on what the use case is for our rewritten binaries (i.e., we have the capability to freely shuffle the program instructions, which would be useful for software diversity).

Once we are done disassembling from offset zero, either because we eventually encountered an illegal instruction or *offset* $\geq$ *length(bytes)*, we start disassembling from offset one. As illustrated in Figure 3, we show a case that the instruction

[1]We call our system MULTIVERSE, since it conservatively assumes that any instruction path that "can happen, does happen" [13].

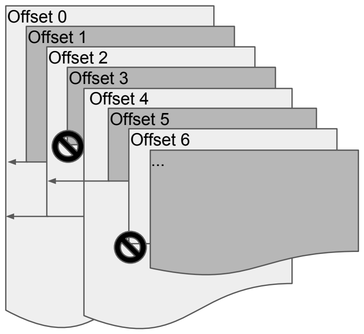

Fig. 3: An illustration of our disassembly strategy.

Algorithm 1: Superset Disassembly

```
input  : empty two-dimensional list instructions
input  : string of raw bytes of text section bytes
output : all disassembled instructions are in instructions
1 for start_offset ← 0 to length(bytes) do
2     offset ← start_offset;
3     while legal(offset) and offset ∉ instructions and
        offset < length(bytes) do
4         instruction ← disassemble(offset);
5         instructions[start_offset][offset] ← instruction;
6         offset ← offset + length(instruction);
7     if offset ∈ instructions then
8         instructions[start_offset][offset] ← "jmp
            offset";
```

sequence starting from offset one shortly encounters an offset that we already encountered in our previous pass starting from offset zero (condition *offset* $\notin$ *instructions* is false for the while loop at line 3 in algorithm 1), so when we rewrite our code we simply insert a jump at the end of that instruction sequence to go to the corresponding instruction from our disassembly from offset zero (line 8). The same thing happens from offsets two and five in Figure 3. However, offsets three and six instead encounter invalid byte sequences that do not correspond to any valid instruction encoding, so we simply stop the disassembly from that offset (condition *legal(offset)* for the while loop at line 3 is falsified).

We can potentially eliminate all code at the end of an instruction sequence ending in illegal code starting from the last CFT instruction (e.g., `jmp` or `ret`). We conservatively include conditional CFT instructions as stopping points, since obfuscated code could include garbage bytes after an always-taken conditional jump. This ensures that the removed bytes are safe to omit, since that subsequence will never be executed unless the original program had a fatal error; if those instructions were executed in the original binary, it would crash when it reached the illegal sequence. This process continues until we reach the end of the text section and have disassembled every possible instruction offset (at which condition $offset < length(bytes)$ for the while loop is false, and $start\_offset = length(bytes)$).

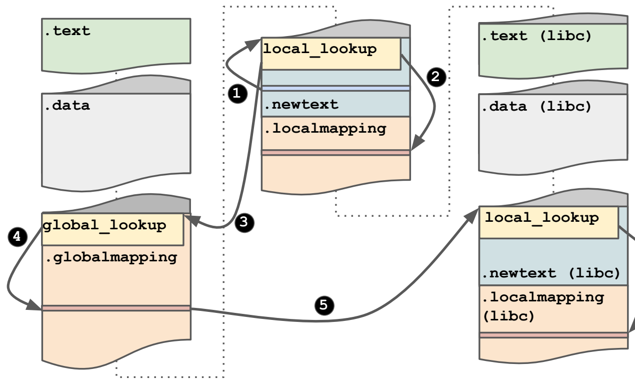

Fig. 4: A mapping lookup example for a rewritten binary dynamically linked with our rewritten `libc`.

B. Mapping Generation

In order to generate the mapping, we retrieve each disassembled instruction from our brute-force disassembler and determine the length of the rewritten instruction that will be in our final binary. It is important to note that since instructions may refer to addresses not yet in our mapping, there is no way to generate the final rewritten bytes in this phase. Therefore, we instead calculate how long each rewritten instruction will be. For most instructions, we make no modifications, and the length is the same.

Specifically, we rewrite all `call`, `jmp`, `jcc`, and `ret` instructions. All the `jcc` instructions involve simply changing the offset for the instruction. However, since we are inserting code, a `jump short` instruction may not have space for a larger offset; if a `jcc` instruction was originally written with a short encoding, we expand it to the longer `jump near` encoding instead, which allows for larger offsets before we know the actual offsets. The other instructions involve adding multiple instructions, so it is important to know how many bytes this adds when we build the final mapping. In practice, we run our rewriter on the instructions with placeholder addresses to substitute for the addresses we do not know, and then retrieve the length of the rewritten instructions in bytes. This strategy of keeping track of instruction length also makes conversion of MULTIVERSE to perform instrumentation quite straightforward; we can simply add the length of inserted instructions to the length of the rewritten instruction.

While we are building our mapping, we maintain a mapping from each old address to the size of the new bytes. When we build the final mapping, we convert the size to the corresponding offset in the new text section. By deferring this to after we disassemble all bytes, we obtain the flexibility to place blocks of new bytes in any order in our mapping as long as we end each block with a `jmp` instruction to the new instruction address corresponding to the next instruction in the old binary, or as long as we split instructions into basic blocks, in which case we would simply need to change direct control flow destinations.

C. Mapping Lookups

For static memory addresses, we modify instructions statically using our mapping offline. However, dynamically computed addresses (C2) require our mapping to be present in the binary at runtime for dynamic lookup and we must use an efficient data structure for reducing runtime overhead. To this end, we generate a flat table of four-byte offsets large enough to have an entry for every byte in the old text section. This allows us to directly index into the table by computing the offset of the old address from the base address of the old text section. For offsets that did not disassemble to a valid instruction, we simply set the entry to `0xffffffff`. For performing lookups in the table, we insert a small assembly function into the binary to look up an address from the old text section and return the corresponding new text section address.

We use the `eax` register as input to pass the old address that we want to look up to the function. Then we use our own PIC (getting the instruction pointer with a `call`) to obtain the offset to the mapping and look up the entry of the old address. If the entry is `0xffffffff`, then the original program had an error and is attempting to jump to an illegal instruction. In such a case, we immediately trigger a segfault by jumping to a `hlt` instruction. If the entry is a valid address, then we return the address in `eax`. If the address is outside the range of the mapping, then the program may be attempting to call a library function, so we pass the address to be resolved by the global lookup function, which we discuss in §III-D.

Figure 4 shows a mapping lookup example for a rewritten binary, showing both its new text and data sections and the text and data sections of a modified `libc`. When a rewritten instruction in `.newtext` requires that we dynamically look up an address, we first call `local_lookup` (❶), which we have placed at the start of the `.newtext` section. This function knows its offset from `.localmapping`, so it can perform a lookup of the destination address (❷). If the address is in the range of the old text section, then it simply returns the new address, our rewritten instruction jumps to that address, and the process is complete. If the address to look up is outside the old text section, we must refer to the global mapping.

D. Global Mapping

Since we are rewriting libraries as well as the original binary, each library has its own local mapping for its new text section. Since the libraries may be loaded dynamically, we must maintain a global mapping between the multiple new text sections that we have generated. As such, we have created a global mapping table and a global lookup function (`global_lookup`) that determines which local lookup function to call to resolve an address.

Function `global_lookup` operates at a page-level granularity. When a library is loaded, its old text section is mapped to one or more pages. The functionality of `global_lookup` is, therefore, to return the address of each library's local lookup function for every page in the library's original text section. In particular, as shown in Figure 4, if a `local_lookup` call does not have an address in its `.localmapping`, then it must call `global_lookup`.

As in the previous example, when an instruction in the `.newtext` section needs to perform a dynamic lookup, it first calls `local_lookup` (❶). If the requested address is that of the `libc` function `qsort`, then it is outside the application's `.text` section (❷). Therefore, it calls `global_lookup` (❸), which finds the entry for `libc`'s `local_lookup` in `.globalmapping` (❹). It calls `libc`'s `local_lookup` (❺), which is then able to find the updated address in its `.localmapping` (❻). Once the new address is found, `libc`'s `local_lookup` returns the address to `global_lookup`, which returns the address to the main binary's `local_lookup`, which finally returns the address to the rewritten instruction.

IV. Rewriting

In our *rewriting* phase, we use our mapping to rewrite all the `call/jmp/ret/jcc` instructions from the old binary in order to preserve the original program's CFTs. When we are rewriting each instruction, we go through the same instructions that we processed during the mapping phase. When we encounter a byte offset that has already been disassembled and rewritten, we insert a `jmp` instruction to the new address of the already rewritten instruction at the end of the current sequence. This allows us to only rewrite the instruction at each offset once.

Rewriting direct CFT instructions (`jcc/jmp/call`). All `jcc` instructions are direct CFTs, so we statically rewrite each `jcc` instruction by changing its offset. In addition, `call` and `jmp` instructions with an immediate operand can also be statically rewritten by changing the offset. However, `jmp short` and `jcc short` instructions only hold a 1-byte displacement, which may not be large enough when we expand our new text section; an instruction's destination may become too distant in our new binary. Therefore, we expand these instructions to their longer encoding (`jmp near` and `jcc near`), allowing for a 4-byte displacement.

Rewriting indirect `jmp/call` instructions. Our static rewriting of indirect control flow instructions implements a dynamic lookup. We must perform a lookup at runtime of the destination address, since the runtime-computed address will point to the old text section. The transformations for `jmp` and `call` instructions are slightly different.

- **`jmp`:** If the instruction is `jmp [target]`, we rewrite it to the following six instructions:
  ```
  mov [esp-32], eax
  mov eax, target
  call lookup
  mov [esp-4], eax
  mov eax, [esp-32]
  jmp [esp-4]
  ```
  We save `eax` to an area outside the stack, move the original target to `eax`, and call `lookup`, which will perform `local_lookup` first and then `global_lookup` if necessary (detailed in §III-C). We then save the result (which was returned in `eax`) outside the stack, restore `eax`, and jump to the new destination. Also note that we are storing `eax` at `esp-32`. This is because the `lookup` function may push up to seven 4-byte values on the stack, and we must store the value outside the reach of the growing stack in order to avoid overwriting the value.
- **`call`:** If the instruction is a `call [target]` instruction, we also push the old return address (the address of the instruction after the old `call`, in order to transparently handle PIC), resulting in these seven instructions:

```
mov [esp-32], eax
mov eax, target
push old_return_address
call lookup
mov [esp-4], eax
mov eax, [esp-28]
jmp [esp-4]
```

Note that we restore `eax` from `esp-28` because we pushed the 4-byte return address on the stack, decrementing `esp` by 4.

Rewriting `ret` instructions. Since we rewrite `call` instructions to push the old return address on the stack, we must perform a runtime lookup of the new return address. In addition, a `ret` instruction may also specify an immediate value specifying a number of additional bytes to pop off the stack, so we may also need to increment `esp` an additional amount. For example, for `ret 8` we need to add 8 to esp. This results in six or seven instructions:

```
mov [esp-28], eax
pop eax
call lookup
add esp,pop_amount; Only add if immediate
mov [esp-4], eax
mov eax, [esp-(32+pop_amount)]
jmp [esp-4]
```

The location where we saved the value of `eax` is relative to `esp`, so we must calculate the offset. However, the calculation `32 + pop_amount` is performed statically. Also note that the preceding examples are for rewriting the original binary. We must insert slightly different code for shared objects because we do not know the runtime base address for a shared object. This means we insert PIC (similar to what we do in the `local_lookup`) in order to push the correct old return value on the stack for each `call` instruction; we must obtain the base address of the old text section of the shared object, which is loaded to a random address. This slightly increases the overhead for shared libraries when compared to the original binary.

V. Implementation

We have implemented Multiverse atop a number of open source binary analysis and rewriting projects. In particular, we used the python bindings for Capstone [1] as our underlying disassembler engine, and we used `pyelftools` [3] to parse the ELF data structures. We used `pwntools` [2] to reassemble the instructions. Additionally, we developed over 3,000 lines of our own python code to implement our algorithm and maintain our data structures, and over 150 lines of assembly, some of which is embedded as string templates in our python code. We also developed over 200 lines of C for the global mapping population function that is run when a rewritten executable starts. Other than the global mapping population function, all of the code that we rewrite or insert into the binary is written in assembly. Note that our system can easily support any disassembler that can perform linear disassembly.

There are several Linux-specific issues that had to be addressed. First of all, the Linux kernel loads a special shared library, the Virtual Dynamic Shared Object (VDSO), which has no actual corresponding `.so` file in the filesystem. One use for this is to run the correct code depending on whether the kernel supports the `sysenter` instruction for syscalls rather than `int 80`. If it does, then control is redirected to the VDSO for each system call. Since our solution requires no changes to the OS, we cannot change the VDSO, so we must instead rewrite the return address before every call into the VDSO. Metadata regarding this is passed to every process in the Auxiliary Vector, which is stored on the stack after the environment variables before application start. We insert code to parse this data structure at the new entry point of the rewritten binary, and we save the address of the VDSO syscall code. Later, whenever the application is about to jump to an address that the local lookup function does not recognize, our global lookup function performs a special check for whether the destination is the VDSO syscall code. If so, we rewrite the return address on the stack from the old address to the new address.

In addition, the dynamic linker is loaded before any other `.so` file, and `libc` calls various functions within it. In order to avoid rewriting the dynamic linker, we instead marked its address range in the global mapping as a special case, allowing us to rewrite return addresses whenever it is called. However, for dynamic function resolution, when an address is first resolved control first goes to the dynamic linker, and then the dynamic linker redirects control directly to the destination, not allowing any of our rewritten code to translate the old address to a new address. We resolve this by setting the environment variable `LD_BIND_NOW` to 1, which forces the loader to resolve all symbols and place their addresses in the GOT before the program starts. This prevents the loader from directly rerouting control to old text sections. This may increase the startup time of a rewritten binary, and symbols may be resolved that are never used, but this does not affect the correctness or safety of the rewritten binary. In fact, disabling lazy loading is part a defense designed to increase the security of the loader mechanism [19].

Finally, we must populate the global mapping when the application starts, as we will not know the actual addresses of each library until they are loaded into memory. Therefore, we insert the global mapping population function, which runs before `_start` in the rewritten binary, to find the address ranges of each library and write them to the global mapping. Later, during execution, the global lookup function uses these mappings to resolve the locations of local lookup functions.

Note that the global mapping only needs to be inserted for executables (i.e., only executables have `.globalmapping`, while all executables and shared libraries have their own `.localmapping`). Specifically, we define the global lookup and mapping to start at the constant address `0x7000000`, which places it below all the sections of most binaries. For unusual binaries with a different layout, we can place it at a different constant address if necessary. Shared libraries will need to call the global lookup function, but since we place it at a fixed address, it does not need to appear in the shared libraries; all the dynamic libraries will call the same global lookup function address because they know it will be mapped there at runtime. This is not restrictive, as we can simply rewrite all the libraries again if we need to change the global mapping to a different address.

VI. Evaluation

In this section, we report our evaluation results. We first report how we evaluate effectiveness in §VI-A, and then

Benchmark	**Dir. Calls**	**Dir. Jumps**	**Ind. Calls**	**Ind. Jumps**	**Cond. Jumps**	**Rets**	**.text** (KB)	**.newtext** (KB)	**Size Inc.** (×)
400.perlbench	30888	24778	3896	4442	126876	22306	1047	5146	12.88
401.bzip2	1100	1050	170	152	7342	874	55	268	70.71
403.gcc	110122	64532	8916	15680	380920	45410	3225	15290	10.32
429.mcf	276	216	44	78	1300	250	12	57	202.98
445.gobmk	23548	14946	3550	3480	117378	20918	1488	6520	5.39
456.hmmer	8020	4942	556	666	28924	4106	277	1279	22.56
458.sjeng	2566	2338	256	658	12236	1570	132	604	36.17
462.libquantum	1094	758	94	146	3376	812	40	181	93.73
464.h264ref	7124	6518	1782	2000	47850	6318	520	2441	16.23
471.omnetpp	33578	10032	3830	1782	51642	14326	635	3029	13.49
473.astar	912	552	162	160	3314	750	39	184	92.52
483.xalancbmk	115154	58678	39392	14630	307122	75674	3850	17369	7.60
libc.so.6	32798	33370	9816	9012	189384	32458	1735	8435	9.77
libgcc_s.so.1	2158	2514	374	484	12862	1740	112	538	9.70
libm.so.6	5450	8870	874	892	21796	7406	277	1268	9.51
libstdc++.so.6	22456	10418	4300	4008	144516	15784	900	4258	9.53

TABLE I: Statistics of MULTIVERSE rewritten binaries and libraries

report the MULTIVERSE performance overhead in §VI-B. For our benchmarks, we used all 12 SPECint 2006 benchmark programs. We also had to rewrite the shared libraries used by the benchmarks. We did not test with the SPECfp benchmarks because we did not focus on rewriting Fortran programs, of which there are several in SPECfp. However, in theory, our rewriter should work on Fortran programs as well. Our test machine runs Ubuntu 14.04.1 LTS, and has an Intel i7-2600 CPU running at 3.40GHz, with 4GiB of RAM.

A. *Effectiveness*

We first demonstrate the effectiveness of MULTIVERSE's implementation by comparing the output of the original and rewritten binaries. By showing that all rewritten binaries produce identical output to the original, we can be confident of the correctness of the implementation of our design. To this end, we executed both the rewritten version and the original version of the corresponding benchmark, and compared their output. All the rewritten binaries run correctly, producing the same output as the original program. We did not attempt to exhaustively run all the branches of the two versions and simply used the same configuration to run them.

Table I summarizes the rewriting statistics, including the binaries and libraries we had to rewrite for the SPEC benchmarks. One interesting detail is the similarity in size overhead for each of the text sections; they all increase in size between 4–5 times. In most x86 binaries, instructions are on average a little over 3 bytes [7], so we speculate that may explain this consistent size increase. For every instruction on average, it may only take an offset of 3-4 bytes to encounter an offset that was already assembled in a previous starting offset (i.e., every 4–5 bytes). We will investigate the implications of this in future work.

The `.newtext` sections shown in the table do not include the local mapping, which is always 4 times larger than the original due to the fact that we must store 4-byte entries for every byte offset in the text section. In addition, every binary also contains the slightly more than 4MB global mapping. It would be possible for us to allocate the global mapping to `.bss` in future work and eliminate this static file overhead, but for now we fill the space with `0xffffffff` bytes. The effect of this can be seen in the last column of Table I. The size overhead for `429.mcf` looks remarkably high because the original binary is very small, and the fixed overhead of the global mapping dominates the rest of the code. Therefore, for large applications, this increase will be less noticeable and the percent increase in size will be much less. This means that as the size of the initial binary increases, size overhead will approach the increase in size of the new text section plus the local mapping, which averages to only around 9 times (4-5 times for `.newtext`, plus 4 times for the local mapping). Also notice that this pattern is demonstrated in the shared libraries we rewrote; since we do not need to include a global mapping in an `.so`, the overhead is lower and more consistent.

Real-World Binaries. We also tested MULTIVERSE with other real-world software to demonstrate its effectiveness. First, we rewrote all the binaries in the GNU Core Utilities, which contain the implementations of utilities found on all Unix-like systems. This provides a diverse set of utility applications. We also tested MULTIVERSE by rewriting a number of other applications, including a graphical browser, web server, and graphical game, among others. In total we rewrote 126 binaries and 77 libraries (the applications shared many common libraries) comprising a total of 54MB. All worked as expected.

B. *Performance*

We also measured the runtime overhead of our rewritten binaries. We ran the SPECint benchmarks 10 times each, both on the original binaries and our rewritten binaries. We took the averages from the benchmark results.

Rewriting Performance. We first rewrote both the benchmarks and all their required shared libraries. As shown in the first bar in Figure 5, a few benchmarks had very high overhead, especially `471.omnetpp` and `483.xalancbmk`. This is because of our very generic handling of control flow during the rewriting. Since we make very few assumptions (discussed in §II-A) about the instructions in a binary, this sometimes results in surprisingly high overhead. Both `471.omnetpp` and `483.xalancbmk` are C++ applications, and therefore we suspect the high overhead results from the use of C++ features

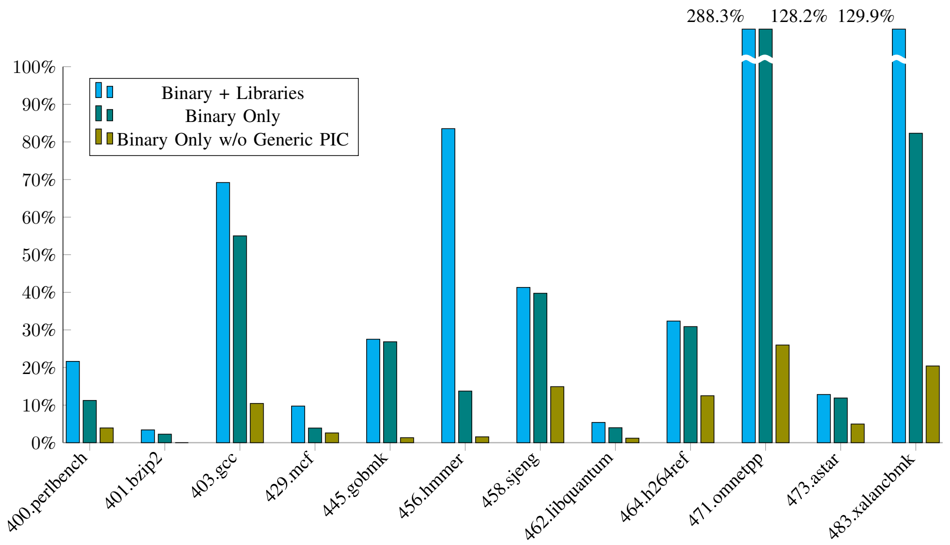

Fig. 5: Percent runtime overhead for each of the reported benchmarks.

that become very expensive after being rewritten. In addition, frequent calls to library functions require a more expensive call to the global lookup, so this may be a factor. However, the other benchmarks all have less than 100% overhead, and most are below 50%. The average runtime overhead when rewriting the main binary and libraries was 60.42%.

We then investigated the impact of some optimizations. In particular, since the contents of libraries are often known, it is acceptable in some use cases to rewrite only the main binary. Therefore, we decided to implement the other approach we discussed for solving C4: only rewriting the main binary (and leaving the libraries unmodified), and treating callbacks as a special case. This requires a list of library functions that take callbacks, since all callback parameters must be rewritten to point to their corresponding addresses in `.newtext`. Since it is difficult to compute this list automatically, we populated it manually. With this approach, we were able to reduce overhead, in some cases significantly (e.g., `456.hmmer` and `471.omnetpp`). The results are shown in the second bar in Figure 5, and average overhead was 34.17%. This also allows the global lookup to be omitted, shrinking the size of the rewritten binary in addition to improving overhead. Rewriting the main binary makes sense in many use cases; sometimes the libraries do not need to be instrumented, or are more trustworthy than the main binary.

In most binaries, PIC uses the `get_pc_thunk` function to get the code address. We found that if we added the extra assumption that code would never attempt to get its own address without using `get_pc_thunk` (a reasonable assumption for well-behaved x86 binaries), we addressed C5 far more efficiently. This is a significant optimization because we no longer need to rewrite all call and return instructions to push and translate old addresses. The effect of this change is clearly shown in the third bar in Figure 5. Average overhead with this optimization was 8.29%. This demonstrates how a few well-chosen assumptions can result in vastly improved performance. Therefore, in the future we can add settings for other common patterns in binaries to improve practical performance when certain properties of a binary are known, while keeping the core rewriter generic enough to handle almost any binary.

It is important to emphasize that writing only the main binary and removing generic PIC rely upon significant assumptions. Specifically, rewriting only the main binary assumes knowledge of all callback arguments for functions, and removing generic PIC assumes that the only PIC in the binary is the thunk. Thus, these two optimizations will not work for certain binaries. However, we demonstrated these optimizations to show the performance improvements they would provide, and to show the potential for adding assumptions in cases in which it is safe to do so. Our core rewriter does *not* make these assumptions.

Instrumentation Performance. Making no changes when rewriting a binary is of limited utility. Binary instrumentation is a much more interesting application that MULTIVERSE facilitates, since we can insert arbitrary code around any existing instruction. We implemented a straightforward instrumentation API to add assembly before any instruction.

Since many tools already perform binary instrumentation, we decided to compare the performance of our instrumented binaries with PIN [26], a widely-used framework for dynamic binary instrumentation from Intel that does not use any heuristics to dynamically disassemble and instrument a binary. We chose PIN due to its popularity and its avoidance of heuristics when rewriting. The goal of this evaluation is to show that the cost of our mapping lookups compares favorably with the state-of-the-art, and thus that instrumenting with our tool is practical.

We decided to compare the overhead of a simple instruction counting instrumentation. Pintool has example tools for instruction counting, and for our evaluation we selected the example Pintool that inserts a call to increment a counter for every instruction. Pin has more efficient strategies for instruction counting, such as inserting instrumentation at the basic block level and incrementing by the number of instructions in each basic block, but our tool does not currently use a basic block abstraction. Therefore we had to compare instrumentation of individual instructions, resulting in higher overhead than would normally

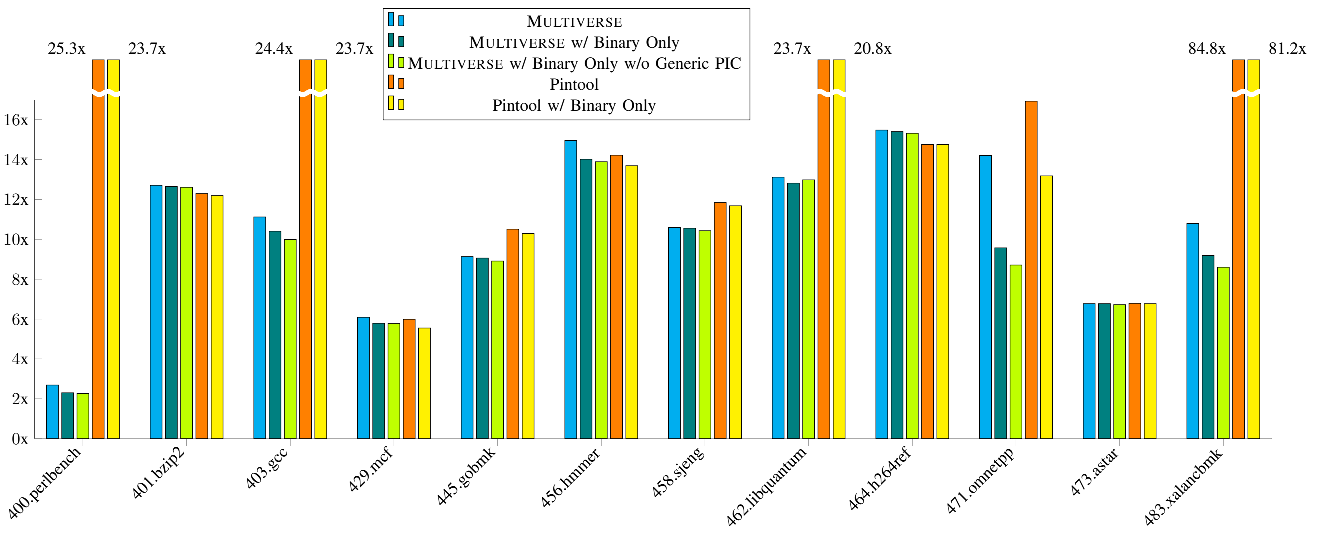

Fig. 6: Runtime overhead (times) for the benchmarks with instruction counting instrumentation.

be expected from an instruction counting Pintool. However, this demonstrates a case in which a static approach is preferable: When each individual instruction requires greater analysis, there is greater advantage in performing the analysis offline.

We wrote our own MULTIVERSE script to insert almost identical assembly to that inserted by Pin for each instruction. (Our insertions are not quite identical to PIN's due to optimizations PIN implements when inserting code, making our inserted assembly slightly less efficient.) We rewrote the SPECint benchmarks with our tool for the main binary with all libraries, as well as with our two optimizations (rewriting only the main binary and rewriting the main binary without generic PIC). Similarly, we used the instruction counting Pintool that instrumented all instructions, and we also created a modified version to only instrument the main binary. We compared the performance by taking the average of running each benchmark 10 times and computing the overhead relative to the results from the unmodified benchmarks.

The results are shown in Figure 6. MULTIVERSE outperformed PIN in many of the benchmarks, and in four cases (`400.perlbench`, `403.gcc`, `462.libquantum`, and `483.xalancbmk`), PIN's performance was substantially worse than MULTIVERSE. This is likely because PIN uses dynamic instead of static instrumentation; any new code encountered by PIN must be analyzed and instrumented on the fly. For benchmarks such as `400.perlbench` or `403.gcc`, there are likely many new paths that are encountered throughout execution, whereas some of the benchmarks with similar performance between MULTIVERSE and PIN may have more loops that PIN can instrument once, leaving only the analysis code to run every subsequent loop. Thus, the additional overhead of Pintool's runtime instrumentation code is probably the cause of such high runtime overhead.

Pintool does run slightly faster for a few benchmarks (e.g. `401.bzip2` and `464.h264ref`). These are likely the result of PIN's superior optimization of inserted code. For the inserted MULTIVERSE assembly, we saved and restored flags before and after every inserted set of instructions. PIN, on the other hand, was able to analyze instructions as it encountered them, inserting code to save and restore flags only when necessary based on analysis of the instrumented code. Therefore, once analysis code is actually inserted, the resulting assembly produced by Pin should be faster than that produced by MULTIVERSE. (Manual analysis of instrumented assembly showed various optimizations performed by PIN.) Since we did no analysis to optimize our inserted code, we have the opportunity to perform static optimizations in the future and increase our performance. Regardless, MULTIVERSE's performance is already promising.

Another interesting result is that the performance improvements from our heuristic optimizations were less significant in most benchmarks when compared to the overhead introduced by the instrumentation. This shows that despite the higher overhead of MULTIVERSE without heuristics, it can be practical in certain instrumentation contexts and can be used for instrumenting binaries that do not obey common assumptions without introducing unacceptable overhead when compared to existing production tools.

VII. SECURITY APPLICATIONS

There is potential for many interesting security applications with MULTIVERSE. Binary rewriting is a foundational technique for increasing security in programs without source available, and MULTIVERSE makes this more practical for arbitrary binaries by making rewriting without heuristics possible. Because of this, and to show the potential of the framework, we used MULTIVERSE to implement a shadow stack.

Shadow stacks are used to protect return addresses on the stack by allocating a separate memory region for the shadow stack, and inserting code that saves return addresses to the shadow stack whenever a function is called. When a function returns, the inserted code either checks whether the return address in the stack and shadow stack match, or simply overwrites the address in the stack with the one stored in the shadow stack. This ensures that an attacker cannot overwrite return addresses in the stack during ROP attacks. Shadow stacks can therefore be considered a form of backward-edge CFI [20]. We implemented an overwriting, no-zeroing, parallel shadow stack [14].

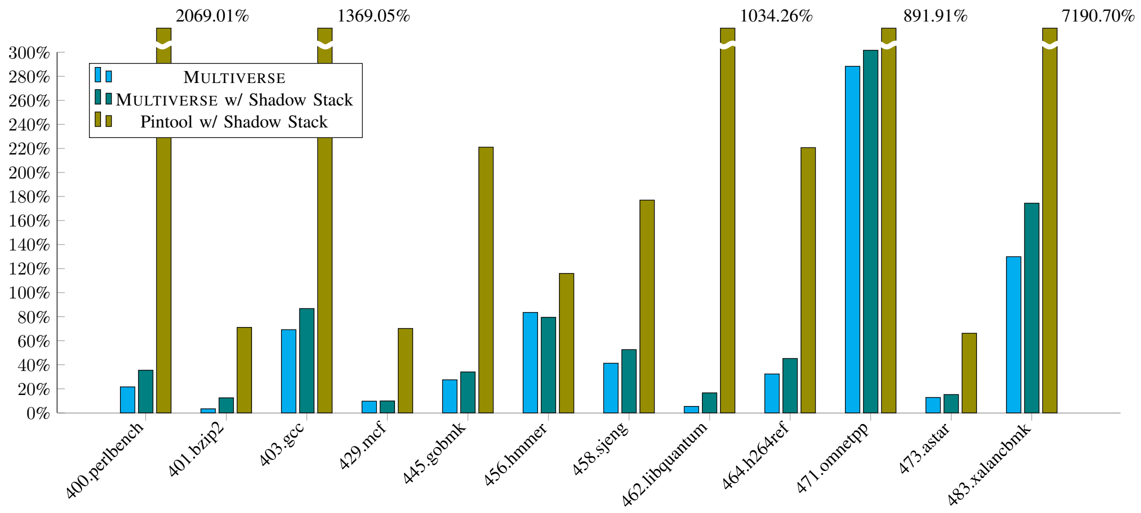

Fig. 7: Percent runtime overhead for the benchmarks instrumented with a shadow stack.

Implementing a shadow stack with our framework was quite straightforward, requiring only the insertion of instructions for every `call` and `ret` instruction and allocating shadow stack memory. Since we rewrite `call` instructions in MULTIVERSE to a `push`/`jmp` pair that pushes the original return address, we must insert our code after the `push` but before the `jmp`. We insert the following two instructions for each `call`, which writes the return address on the top of the stack into the parallel shadow stack:

```
pop [esp + (shadow_stack_offset - 4)]
sub esp, 4
```

MULTIVERSE rewrites `ret` instructions into `pop`/`jmp` instructions, in which the return address at the top of the stack is popped and passed to the lookup function to determine the rewritten jump target. Therefore, we simply insert two instructions directly before the rewritten `ret` code to overwrite the return address on the stack with the corresponding shadow stack contents (hence an *overwriting* shadow stack):

```
add esp, 4
push [esp + (shadow_stack_offset - 4)]
```

We rewrote the SPECint benchmarks and their libraries without either of our optimizations, because the shadow stack does not work when only rewriting the main binary. Some functions in `libc` call code in the main binary, and since `libc` is not rewritten when we use our optimizations, it does not push the return address on the shadow stack. Therefore, when the main binary returns to `libc`, there is no entry in the shadow stack and the program crashes. This is not a limitation of our framework, but rather an implementation challenge for shadow stacks if one does not intend to instrument all calls. Therefore, for our shadow stack proof-of-concept, we focus only on our general approach of rewriting everything.

Shadow Stack Performance. We ran the SPECint benchmarks 10 times and computed the overhead relative to the results of the unmodified benchmarks. The results are shown in the second column of Figure 7. Performance results are similar to our framework with no instrumentation, with an average increase of 11.64% when compared to the overhead in the first column (the overhead of MULTIVERSE with no instrumentation, which is the same as the first column in Figure 5). The benchmark with the highest increase over no instrumentation was `483.xalancbmk`, with an increase of 44.49%, which was also the benchmark with the highest overhead for all shadow stack implementations in [14]. Several other benchmarks have a very low increase in overhead, such as `429.mcf` (0.19%) and `473.astar` (2.44%); and one benchmark, `456.hmmer`, was in fact faster (-4.02%). Speed improvements are reported for several benchmarks in the reference implementation [14] as well, so this is not too surprising.

The overhead difference over our baseline rewriter for SPECint is more significant than the overhead shown in [14]; however, the data presented there is for SPECint benchmarks compiled with `-O3`, which the paper claims is marginally faster than the default `-O2` optimization level we used. In addition, the reference implementation [14] assumes knowledge of the correct assembly, and inserts the instrumentation before the object files are assembled, yielding the lowest instrumentation cost possible. Finally, that implementation instruments function prologues instead of call instructions; we instrument call instructions because we do not have reliable information about function entry points, which may result in more instrumentation (and less code locality). These differences likely contribute to the higher increase in overhead for MULTIVERSE when compared to a near-optimal shadow stack implementation.

We also decided to implement the same type of shadow stack in PIN and compare the performance. We attempted to implement the shadow stack as close as possible in behavior to our implementation in MULTIVERSE. However, the inserted code was written in C++ instead of assembly, and we used the PIN API to access the value of `esp` since it has a different value when our inserted analysis code accesses it directly. The results are shown as the third column in Figure 7, and the performance is significantly slower than MULTIVERSE's implementation. While we do believe that the PIN implementation can be optimized, the PIN API appears to make it difficult to directly access register values in an original instruction's context as efficiently as assembly code directly inserted before that instruction. This may be one reason for the significant overhead.

VIII. Limitations and Future Work

Our current implementation of Multiverse is merely a proof of concept. While our general approach can support x64 applications and other operating systems, our prototype currently supports only x86 Linux ELF executables and `.so` files. In this section, we discuss some of the limitations of our system and outline how we plan address them in future work.

Supporting x64 applications mainly entails changing the assembly language of the rewritten instructions and lookup functions, but there are a few more significant changes needed. For example, in x64, instructions can directly access the contents of the `rip` register, so PIC does not require thunks. This changes the way PIC must be handled by the rewriter, since all references to `rip` must be modified to accommodate the position of the new code. Our progress on x64 support is almost complete.

Our system has several aspects that can be optimized. Since our priority was generality, we do not address special situations in which we could optimize away some of our more expensive runtime computations, especially when rewriting both `.so` files and the main binary. Meanwhile, in optimizing our mapping data structures for speed, we have made the tradeoff of additional space overhead. As shown by the optimizations we have already performed, we expect that future refinements will yield gains in both speed and size.

In addition, while we have developed a fundamental building block for rewriting binaries, we have demonstrated only one concrete application. However, as discussed at the end of §VI-B, our prototype instrumentation framework using Multiverse facilitates instrumentation, such as counting the number of instructions executed. In addition, we have demonstrated a practical example of instrumentation with a shadow stack. We will expand on this instrumentation ability and its use cases in future work.

Other applications of Multiverse, such as binary hardening and other transformations that alter some of the original instructions in a binary, may require minor changes to be compatible with our mapping and lookup strategy, but Multiverse should not impose significant challenges to implementing known techniques such as SFI. In fact, the mapping could potentially be used to assist enforcement for some security policies. We are working on applications such as these for future work.

IX. Related Work

Rewriting binary code can be dated back to the late 1960s [16], when it was first used for flexible performance measurement. Later in the late 1980s and early 1990s, rewriting binaries for RISC architectures (e.g., Alpha, SPARC, and MIPS), in which code is well-aligned and instructions have a fixed length, became quite popular in applications such as instrumentation (e.g., PIXIE [12], ATOM [38]), performance measurement and optimization (e.g., QPT [23] and EEL [24]), and architecture translation [35]. In contrast to RISC architectures, rewriting binaries for CISC such as x86 is much more challenging. We traced the earliest attempt to rewrite x86 binaries to Etch [32]. In the following, we review prior efforts of x86 static binary rewriting and compare them with Multiverse. We here omit a survey of dynamic binary rewriting (e.g., Pin [26]), since it encounters a different set of challenges.

Targeting instrumentation, profiling, and optimization, Etch [32] made a first step towards rewriting arbitrary Win32/x86 binaries, potentially without any relocation entries or debugging symbols. However, the implementation details of Etch are not open and it is very likely that Etch used heuristics for disassembling, recognizing static memory addresses, handling callbacks and PIC. Instead of rewriting arbitrary x86 binaries, SASI [17] focused on rewriting only `gcc` produced binaries for SFI [41]. Similarly, PittSFIeld [27] and Native Client (NaCl) [49] also require cooperation from compilers.

Unlike stripped binaries, object code contains rich information, such as where the code is located, and which data contains static memory addresses. Therefore, a number of efforts focused on rewriting x86 object code without heuristics. Plto [33] and Diablo [31] are both examples that target program optimization and profiling. With debugging symbols (which is almost as informative as source code), Vulcan [37] (and Pebil [25]) can correctly rewrite x86 binaries without using any heuristics. Vulcan was later used in two security systems: CFI [4] and XFI [18]. Pebil was later extended and used in StackArmor [11] for stack memory protection.

Bird [29] is the first system that targets COTS binary rewriting. In particular, Bird first uses compiler idioms and various assumptions to disassemble as many instructions as possible, and it further improves results with an on-demand runtime disassembler. Meanwhile, with a runtime exception mechanism, Bird can also safely handle PIC and callbacks. In addition to regular applications such as profiling, Bird has been used to harden a binary for foreign code detection.

SecondWrite [30] lifts the disassembled code into LLVM IR [5] to optimize the original binaries for better runtime performance by leveraging the strength of LLVM optimization. While SecondWrite can rewrite a binary without supplementary information [36], it still uses heuristics for binary code disassembling and callback handling, and it also does not handle PIC in its current implementation [36]. It also inherits limitations of the LLVM IR, which cannot easily encode certain native code programs containing instruction sequences that LLVM compilers do not generate.

DynInst [6] is a framework that supports both static and dynamic binary instrumentation. While the published literature on DynInst is unclear on whether it supports PIC and callbacks, our investigations showed that it uses heuristics to handle PIC based on particular patterns. DynInst has been improved and used in various applications, such as performance studies and binary hardening (e.g., PathArmor [39], TypeArmor [40], and CodeArmor [10]).

Stir [46] and Reins [47] preserve the original program's functionality by tracking basic block addresses and then randomizing them or in-lining them with security logic (respectively) to mitigate attacks. Both systems rely on a *shingled disassembly* strategy that resembles our superset disassembly, but that applies imperfect machine learning heuristics to optimize rewritten code

Systems	Year	w/o Relocation	w/o (Debugging) Symbols	w/o Heuristics for Static Address	w/o Heuristics for PIC	w/o Heuristics for Callbacks	w/o Heuristics for Disassemble	Instrumentation	Profiling	Optimization	Binary Code Hardening	Control Flow Integrity	Binary Code Reuse
ETCH [32]	1997	✓	✓	✗	✗	✗	✗	✓	✓	✓	✗	✗	✗
SASI [17]	1999	✗	✗	✓	✓	✓	✓	✗	✗	✗	✓	✗	✗
PLTO [33]	2001	✗	✗	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗
VULCAN [37]	2001	✓	✗	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗
DIABLO [31]	2005	✗	✗	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗
CFI [4]	2005	✓	✗	✓	✓	✓	✓	✗	✗	✗	✓	✓	✗
XFI [18]	2006	✓	✗	✓	✓	✓	✓	✗	✗	✗	✓	✗	✗
PITTSFIELD [27]	2006	✗	✗	✓	✓	✓	✓	✗	✗	✗	✓	✗	✗
BIRD [29]	2006	✓	✓	✗	✓	✓	✗	✓	✓	✓	✓	✗	✗
NACL [49]	2009	✗	✗	✓	✓	✓	✓	✗	✗	✗	✓	✗	✗
PEBIL [25]	2010	✗	✗	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗
SECONDWRITE [30]	2011	✓	✓	✓	✗	✗	✗	✓	✓	✓	✓	✗	✗
DYNINST [6]	2011	✓	✓	✗	✗	✓	✗	✓	✓	✓	✓	✓	✗
STIR/REINS [46], [47]	2012	✓	✓	✓	✗	✗	✓	✗	✗	✗	✓	✓	✗
CCFIR [51]	2013	✗	✓	✓	✓	✗	✗	✗	✗	✗	✓	✓	✗
BISTRO [15]	2013	✓	✓	✓	✗	✗	✗	✗	✗	✗	✓	✗	✓
BINCFI [53]	2013	✓	✓	✓	✓	✓	✗	✗	✗	✗	✓	✓	✗
PSI [52]	2014	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✗
UROBOROS [44]	2016	✓	✓	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓
RAMBLR [42]	2017	✓	✓	✓	✗	✗	✗	✓	✓	✓	✓	✓	✓
MULTIVERSE	2018	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

TABLE II: Comparison w/ existing x86 binary rewriters.

sizes [48]. Subsequent work has leveraged these foundations to implement Opaque Control-flow Integrity [28]. Callback support for these systems has recently been automated by the advent of Object Flow Integrity [45], but PIC support remains heuristic-based. Lookup table implementation in these systems is also less robust than MULTIVERSE, since it adopts an encoding that assumes iCFT targets are at least 5 bytes apart—an assumption violated by many COTS binaries.

CCFIR [51] leverages relocation information to disassemble program code. While CCFIR does not have any issues in recognizing addresses or handling PIC due to the use of relocation information, it still uses heuristics for disassembly and handling callbacks. Also assuming correct disassembly, BISTRO [15] rewrites binaries for binary code hardening and reuse [8], [50]. It also uses heuristics to handle PIC and callbacks.

To our knowledge, BINCFI [53] is the first system that can safely handle static memory addresses, PIC, and callbacks. While it also has attempts for better disassembly, BINCFI still cannot handle interleaved code and data well. PSI [52] makes BINCFI more general as a framework for binary rewriting in various other applications such as profiling. UROBOROS [43] makes binary disassembly reassemblable by using the same disassembling algorithm from BINCFI, but it still uses a number of heuristics to differentiate memory addresses and constant integers when relocating a binary. The recent extension of UROBOROS [44] has been made to be a more general static binary instrumentation framework.

Most recently, RAMBLR [42], built atop angr [34], has attempted to remove the static memory heuristics used by UROBOROS [43]. For example, UROBOROS [43] assumes that code pointers in the data section are n-byte aligned and only point to function entry points or jump table entries. RAMBLR [42] instead performs localized data flow and value-set analysis to recognize pointers and integers. Note that RAMBLR and angr do not aim to solve other disassembly problems such as handling PIC and callbacks without using heuristics, which is the core focus of MULTIVERSE.

Compared to all of the existing works, we notice that MULTIVERSE is the first system that is founded on an approach starting with no heuristics in x86 COTS binary rewriting. It can be used in all existing applications, such as profiling, optimization, binary code hardening, CFI, and binary code reuse. A summary of our comparison is presented in Table II.

X. CONCLUSION

We have presented MULTIVERSE, the first static binary rewriting tool that can correctly rewrite an x86 COTS binary without using heuristics. It consists of two fundamental techniques: superset disassembly that completely disassembles the binary code into a superset of instructions that contain all legal instructions, and instruction rewriting that is able to relocate instructions to any other location by interposing all the iCFTs and redirecting them to the correct new addresses. We have implemented MULTIVERSE atop a number of binary analysis and rewriting tools (e.g., CAPSTONE, `pyelftools` and `pwntools`), and tested with SPECint 2006. Our experimental results show that MULTIVERSE is able to rewrite all of the testing binaries and the runtime overhead for the new rewritten binaries is still reasonable. Comparison with another solution without heuristics (dynamic instrumentation) also shows that the static instrumentation enabled by MULTIVERSE can achieve better average performance, and our shadow stack implementation shows that MULTIVERSE can be used for actual security applications.

XI. AVAILABILITY

The source code of MULTIVERSE is made available at github.com/utds3lab/multiverse.

REFERENCES

[1] Capstone: The ultimate disassembler. http://www.capstone-engine.org/.

[2] Pwntools. https://github.com/Gallopsled/pwntools.

[3] Pyelftools. https://github.com/eliben/pyelftools.

[4] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. Information and System Security (TISSEC)*, 13(1), 2009.

[5] 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 *Proc. 8th ACM European Conf. Computer Systems (EuroSys)*, pages 295–308, 2013.

[6] A. R. Bernat and B. P. Miller. Anywhere, any-time binary instrumentation. In *Proc. 10th ACM SIGPLAN-SIGSOFT Work. Program Analysis for Software Tools (PASTE)*, pages 9–16, 2011.

[7] E. Blem, J. Menon, and K. Sankaralingam. Power struggles: Revisiting the RISC vs. CISC debate on contemporary ARM and x86 architectures. In *Proc. 19th IEEE Int. Sym. High Performance Computer Architecture (HPCA)*, pages 1–12, 2013.

[8] J. Caballero, N. M. Johnson, S. McCamant, and D. Song. Binary code extraction and interface identification for security applications. In *Proc. 17th Annual Network & Distributed System Security Sym. (NDSS)*, 2010.

[9] J. Caballero and Z. Lin. Type inference on executables. *ACM Computing Surveys*, 48(4):65:1–65:35, 2016.

[10] X. Chen, H. Bos, and C. Giuffrida. CodeArmor: Virtualizing the code space to counter disclosure attacks. In *Proc. 2nd IEEE Sym. Security and Privacy (EuroS&P)*, pages 514–529, 2017.

[11] X. Chen, A. Slowinska, D. Andriesse, H. Bos, and C. Giuffrida. StackArmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries. In *Proc. 22nd Annual Network & Distributed System Security Sym. (NDSS)*, 2015.

[12] F. C. Chow, M. I. Himelstein, E. Killian, and L. Weber. Engineering a RISC compiler system. In *Proc. 31st IEEE Computer Society Int. Conf. (COMPCON)*, pages 132–137, 1986.

[13] B. Cox and J. Forshaw. *The Quantum Universe: (And Why Anything That Can Happen, Does)*. Da Capo Press, 2012.

[14] T. H. Dang, P. Maniatis, and D. Wagner. The performance cost of shadow stacks and stack canaries. In *Proc. 10th ACM Sym. Information, Computer and Communications Security (AsiaCCS)*, pages 555–566, 2015.

[15] Z. Deng, X. Zhang, and D. Xu. Bistro: Binary component extraction and embedding for software security applications. In *Proc. 18th European Sym. Research in Computer Security (ESORICS)*, pages 200–218, 2013.

[16] P. Deutsch and C. A. Grant. A flexible measurement tool for software systems. In *Proc. IFIP Congress, Volume 1*, pages 320–326, 1971.

[17] Ú. Erlingsson and F. B. Schneider. SASI enforcement of security policies: A retrospective. In *Proc. New Security Paradigms Work. (NSPW)*, pages 87–95, 1999.

[18] Ú. Erlingsson, S. Valley, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *Proc. 7th USENIX Sym. Operating Systems Design and Implementation (OSDI)*, 2006.

[19] A. D. Federico, A. Cama, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. How the ELF ruined Christmas. In *Proc. 24th USENIX Security Sym.*, pages 643–658, 2015.

[20] Y. Gu, Q. Zhao, Y. Zhang, and Z. Lin. PT-CFI: Transparent backward-edge control flow violation detection using Intel processor trace. In *Proc. 7th ACM Conf. Data and Application Security and Privacy (CODASPY)*, pages 173–184, 2017.

[21] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *Proc. 13th USENIX Security Sym.*, 2004.

[22] E. Ladakis, G. Vasiliadis, M. Polychronakis, S. Ioannidis, and G. Portokalidis. GPU-Disasm: A GPU-based x86 disassembler. In *Int. Information Security Conf.*, pages 472–489, 2015.

[23] J. R. Larus and T. Ball. Rewriting executable files to measure program behavior. *Software Practice and Experience*, 24(2):197–218, 1994.

[24] J. R. Larus and E. Schnarr. EEL: Machine-independent executable editing. In *Proc. 16th ACM Conf. Programming Language Design and Implementation (PLDI)*, pages 291–300, 1995.

[25] M. A. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely. PEBIL: Efficient static binary instrumentation for Linux. In *Proc. IEEE Int. Sym. Performance Analysis Systems and Software (ISPASS)*, pages 175–183, 2010.

[26] C.-K. 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 *Proc. 26th ACM Conf. Programming Language Design and Implementation (PLDI)*, pages 190–200, 2005.

[27] S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In *Proc. 15th USENIX Security Sym.*, 2006.

[28] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz. Opaque control-flow integrity. In *Proc. 22nd Annual Network & Distributed System Security Sym. (NDSS)*, 2015.

[29] S. Nanda, W. Li, L.-C. Lam, and T. Chiueh. BIRD: Binary interpretation using runtime disassembly. In *Proc. 4th IEEE/ACM Int. Sym. Code Generation and Optimization (CGO)*, pages 358–370, 2006.

[30] P. O'Sullivan, K. Anand, A. Kotha, M. Smithson, R. Barua, and A. D. Keromytis. Retrofitting security in COTS software with binary rewriting. In *Proc. 26th IFIP TC Int. Information Security Conf. (SEC)*, pages 154–172, 2011.

[31] L. V. Put, D. Chanet, B. D. Bus, B. D. Sutter, and K. D. Bosschere. DIABLO: A reliable, retargetable and extensible link-time rewriting framework. In *Proc. 5th IEEE Int. Sym. Signal Processing and Information Technology (ISSPIT)*, pages 7–12, 2005.

[32] 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 *Proc. USENIX Windows NT Work.*, pages 1–7, 1997.

[33] B. Schwarz, S. Debray, G. Andrews, and M. Legendre. Plto: A link-time optimizer for the Intel IA-32 architecture. In *Proc. Work. Binary Translation (WBT)*, 2001.

[34] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna. Firmalice – automatic detection of authentication bypass vulnerabilities in binary firmware. In *Proc. 22nd Annual Network & Distributed System Security Sym. (NDSS)*, 2015.

[35] R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, and S. G. Robinson. Binary translation. *Communications ACM (CACM)*, 36(2):69–81, 1993.

[36] M. Smithson, K. Elwazeer, K. Anand, A. Kotha, and R. Barua. Static binary rewriting without supplemental information: Overcoming the tradeoff between coverage and correctness. In *Proc. IEEE 20th Working Conf. Reverse Engineering (WCRE)*, pages 52–61, 2013.

[37] A. Srivastava, A. Edwards, and H. Vo. Vulcan: Binary transformation in a distributed environment. Technical Report MSR-TR-2001-50, Microsoft Research, 2001.

[38] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proc. 15th ACM Conf. Programming Language Design and Implementation (PLDI)*, pages 196–205, 1994.

[39] V. van der Veen, D. Andriesse, , E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida. Practical context-sensitive CFI. In *Proc. 22nd ACM Conf. Computer and Communications Security (CCS)*, pages 927–940, 2015.

[40] V. van der Veen, E. Göktaş, M. Contag, A. Pawlowski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *Proc. 37th IEEE Sym. Security & Privacy (S&P)*, pages 934–953, 2016.

[41] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proc. 14th ACM Sym. Operating System Principles (SOSP)*, pages 203–216, 1993.

[42] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel, and G. Vigna. Ramblr: Making reassembly great again. In *Proc. 24th Annual Network & Distributed System Security Sym. (NDSS)*, 2017.

[43] S. Wang, P. Wang, and D. Wu. Reassembleable disassembling. In *Proc. 24th USENIX Security Sym.*, pages 627–642, 2015.

[44] S. Wang, P. Wang, and D. Wu. UROBOROS: Instrumenting stripped binaries with static reassembling. In *Proc. IEEE 23rd Int. Conf. Software Analysis, Evolution, and Reengineering (SANER)*, pages 236–247, 2016.

[45] W. Wang, X. Xu, and K. W. Hamlen. Object flow integrity. In *Proc. 24th ACM Conference on Computer and Comunications Security (CCS)*, pages 1909–1924, 2017.

[46] R. Wartell, V. Mohan, K. Hamlen, and Z. Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proc. 19th ACM Conf. Computer and Communications Security (CCS)*, pages 157–168, 2012.

[47] R. Wartell, V. Mohan, K. Hamlen, and Z. Lin. Securing untrusted code via compiler-agnostic binary rewriting. In *Proc. 28th Annual Computer Security Applications Conf. (ACSAC)*, pages 299–308, 2012.

[48] R. Wartell, Y. Zhou, K. W. Hamlen, and M. Kantarcioglu. Shingled graph disassembly: Finding the undecideable path. In *Pacific-Asia Conf. Knowledge Discovery and Data Mining (PAKDD)*, pages 273–285, 2014.

[49] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. In *Proc. 30th IEEE Sym. Security & Privacy (S&P)*, pages 79–93, 2009.

[50] J. Zeng, Y. Fu, K. Miller, Z. Lin, X. Zhang, and D. Xu. Obfuscation-resilient binary code reuse through trace-oriented programming. In *Proc. 20th ACM Conf. Computer and Communications Security (CCS)*, pages 487–498, 2013.

[51] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity and randomization for binary executables. In *Proc. 34th IEEE Sym. Security & Privacy (S&P)*, pages 559–573, 2013.

[52] M. Zhang, R. Qiao, N. Hasabnis, and R. Sekar. A platform for secure static binary instrumentation. In *Proc. 10th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environments (VEE)*, pages 129–140, 2014.

[53] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *Proc. 22nd USENIX Security Sym.*, pages 337–352, 2013.