Towards Sound Reassembly of Modern x86-64 Binaries

Hyungseok Kim
The Affiliated Institute of ETRI
Daejeon, Korea
hskim@nsr.re.kr

Soomin Kim
KAIST
Daejeon, Korea
soomink@kaist.ac.kr

Sang Kil Cha
KAIST
Daejeon, Korea
sangkilc@kaist.ac.kr

Abstract

Reassembly is a promising approach to transparently rewrite binaries without source code. However, sound symbolization remains an open problem as it requires precise identification of all memory references in the binary. In this paper, we systematically study the requirements for sound reassembly of modern x86-64 binaries, and present a novel approach to reassembly that symbolizes all memory references without affecting the original semantics. The key insights are twofold: (1) we find that Control-flow Enhancement Technology (CET), which has increasingly become the default setting for major Linux distributions, adds a unique property to binaries that can be leveraged to precisely symbolize dynamically computed pointers, and (2) we consider a superset of all possible memory references for symbolization by over-approximating indirect branch targets. With these insights, we design and implement a novel reassembler, named SURI, and show its effectiveness on 9,600 real-world binaries.

CCS Concepts: • **Software and its engineering → Software reverse engineering**; • **Security and privacy → Software reverse engineering**.

Keywords: reassembly; binary rewriting; reverse engineering; binary hardening; binary analysis; superset CFG

ACM Reference Format:
Hyungseok Kim, Soomin Kim, and Sang Kil Cha. 2025. Towards Sound Reassembly of Modern x86-64 Binaries. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3676641.3716026

1 Introduction

Binary rewriting is essential for enhancing the security of Commercial-Off-The-Shelf (COTS) applications where source code is unavailable. Unlike dynamic binary instrumentation approaches [6, 9, 9, 31, 35, 37], binary rewriting is a static approach that modifies the binary code before execution, causing negligible runtime overhead compared to dynamic approaches. Therefore, it is desirable for security applications that require high performance, such as Control-Flow Integrity (CFI) [1, 20, 55, 57].

Reassembly [22, 47, 48, 52] is a promising approach to binary rewriting where the binary code is transformed into relocatable assembly code, allowing effortless addition of instrumentation and patching. Since it does *not* require any runtime support, such as table lookups [5, 42, 51, 56] and detouring [8, 15, 18, 24, 29] it can achieve significantly better performance than other binary rewriting techniques.

Unfortunately, reassembly has never been a solved problem [26]. The main difficulty lies in identifying and translating numbers into symbolic labels, which is often referred to as a *symbolization* process. For example, given an x86-64 instruction `push 0x12345678`, one should be able to decide whether the immediate is an address or a simple literal, and transform it into a symbolic label if it is an address in order to make the disassembled code *relocatable*.

Recent studies [17, 46, 52] have shown potential for *sound* reassembly when the target binary is a Position-Independent Executable (PIE). Indeed, one can easily distinguish an address from a numeral in a PIE binary as every absolute address is tagged with relocation information, which is used by the loader to relocate the binary at runtime.

However, sound symbolization (and, thus, sound reassembly) is still far from being solved even for PIE binaries as noted by Kim *et al.* [26], although previous studies inaccurately claim the soundness of their tools [17, 46]. The key challenge is that compilers often use complex symbolic expressions to represent addresses, which are inherently difficult to correctly symbolize. For instance, a PC-relative operand is often represented as a combination of a label and a numeral, e.g., "`RIP + label`$_1$` + 0x42`", but the operand will contain only a single numeral after compilation, e.g., `RIP + 0x10042`, which could potentially be misinterpreted as an address of another data or code block.

In this paper, we subdivide the types of symbolic labels identified by Kim *et al.* [26] into seven categories by solely focusing on x86-64 binaries. We then analyze how existing symbolization techniques address each category of symbolic labels, or fail to do so. Next, we identify two remaining (and less explored) challenges that need to be addressed for sound reassembly: (1) reliable identification of dynamically computed pointers, and (2) complete recovery of indirect branches. Consequently, we present novel and practical solutions to handle both challenges.

Our approach is based on the following two observations. First, modern Intel binaries, where Control-flow Enforcement Technology (CET) is enabled, have unique code patterns that help symbolize dynamically computed pointers, which are otherwise difficult. Note that Intel introduced CET in 2016 to mitigate control-flow hijack exploits at the hardware level [44], and it has been officially supported since the 11th generation. Furthermore, most Linux distributions today release their binaries by enabling both CET and PIE as we show in §2.3. Second, we can fully recover nodes and edges in a CFG by over-approximating possible indirect branch targets of it, which we refer to as a *superset CFG*. Although a superset CFG may contain bogus nodes and edges, we devise a novel way to handle them without affecting the correctness of the reassembled binary.

We design and implement **Su**perset CFG-based **R**eliable **I**nstrumentation framework, named SURI, to demonstrate the feasibility of sound reassembly for CET-enabled x86-64 PIE binaries. We evaluate SURI on the largest reassembly benchmark to date, consisting of 9,600 real-world binaries, and show that SURI can soundly reassemble all of them. Furthermore, we show that the reassembled binaries incur negligible overhead when compared to the original binaries (0.2% on average). In summary, our contributions are:

- We systematically analyze the challenges in reassembling CET-enabled x86-64 PIE binaries, and identify the key requirements for sound reassembly.
- We propose a novel way to reliably symbolize pointers by using the characteristics of CET-enabled binaries.
- We propose a novel disassembly technique that recursively recovers instructions and constructs a superset CFG that includes all possible indirect edges.
- We demonstrate SURI, a symbolization-based binary rewriting framework that enables reliable instrumentation of CET-enabled x86-64 binaries, and show that the rewritten binaries run with negligible overhead.
- We publicize our framework to foster future research: https://github.com/SoftSec-KAIST/SURI.

2 Background And Motivation

In this section, we start by defining our problem scope. We then introduce Intel CET and discuss its prevalence. Next, we describe how compilers generate symbolic labels for x86-64 PIE binaries, and discuss why recovering them is challenging. Finally, we motivate our approach by presenting symbolization errors found in the SOTA reassemblers.

2.1 Problem Scope

We focus on reassembling binaries that are compiled from C/C++/Fortran[1] source code but do not consider hand-written assembly code nor self-modifying binaries. That is, we assume that our target programs only perform operations specified by the C/C++/Fortran language standard. For example, we assume that one will *not* write a program that performs unsafe function pointer arithmetic, which is undefined behavior according to the C standard. Finally, we only consider CET-enabled x86-64 PIE binaries as our target, which are prevalent in modern Linux distributions.

2.2 Intel CET

Intel Control-flow Enforcement Technology (CET) enforces Control-Flow Integrity (CFI) at a hardware level. Specifically, Intel CET includes two different memory protection mechanisms, Shadow Stacks (SHSTK) and Indirect Branch Tracking (IBT), to protect backward and forward indirect control flows, respectively. First, SHSTK stores return addresses in a separate memory region, called the shadow stack, to protect against stack smashing attacks. Second, IBT verifies that every indirect branch jumps to a predefined valid location, marked by so-called `endbr` instructions.

A recent study [27] showed that the use of `endbr` instructions in modern CET-enabled binaries has a significant implication for binary analysis, particularly for identifying function entry points. In this paper, we build upon this observation and propose a novel solution to address the symbolization challenges in CET-enabled binaries.

2.3 Prevalence of Intel CET-enabled Binaries

Linux distributions have been actively adopting Intel CET to enhance the security of their systems. Particularly, Linux enables IBT and SHSTK support since kernel v6.2 [32] and v6.6 [40], respectively.

To assess the prevalence of Intel CET-enabled binaries in Linux distributions, we examined popular Linux distributions, including Arch Linux 2024.05.01, CentOS 8, Fedora 40, and Ubuntu 24.04. We first checked what kind of default compilation flags are used for building packages in each distribution. We found that *all* distributions have utilized the `-fcf-protection` option for building their packages [3, 10, 21, 45], meaning that CET is enabled by default.

We also downloaded their official Docker images and investigated executable binaries in `/bin`, `/sbin`, `/usr/bin` and `/usr/sbin`. We observed that 99.9% of the binaries in the Docker image contain `IBT` and `SHSTK` properties in the `.note.gnu.property` section, indicating that they are indeed CET-enabled binaries. It is worth noting that there were even more CET-enabled binaries than PIE-enabled binaries; 99.7% of the binaries were both CET-enabled and PIE-enabled. This observation suggests that our approach is applicable to a wide range of modern Linux systems.

While CET is increasingly being adopted in Linux distributions, it is not yet widely supported in other operating systems, such as Windows and macOS. Therefore, our approach is mainly applicable to Linux systems at the moment.

[1]SPEC CPU2006 and SPEC CPU2017, which are two of the benchmarks we used, contain programs written in Fortran.

Table 1. Seven different types of symbolic labels appeared in x86-64 assembly code generated by compilers, their characteristics, and the impact of suggested solutions. Solution ① and ② are existing ones, and ③ and ④ are proposed in this paper. ① is focusing on PIEs†, ② is fixing the layouts of data sections‡, ③ is CET-based pointer repairing, and ④ is superset symbolization.

ID	Representation	Format	Location	Ptr. Type	Without Solution	Impact of Solutions				Example Code (ℓ represents a label)
						Previous Solutions		Our Solutions		
						①	① + ②	① + ② + ③	① + ② + ③ + ④	
S1	Absolute Address	(**F1**) Label	Code/Data	Code/Data	✗	✓	✓	✓	✓	`.quad ℓ`
S2		(**F2**) Label ± Const.	Code/Data	Code/Data	✗	✗	✗	✓	✓	`.quad ℓ + 42`
S3		(**F3**) Label − Label	Code/Data	Data	✗	✗	✓	✓	✓	`.long ℓ1 − ℓ2`
S4			Data	Code	✗	✗	✗	✗	✓	
S5	Relative Offset	(**F1**) Label	Code	Code	✓	✓	✓	✓	✓	`jmp ℓ`
S6		(**F4**) `RIP` + Label	Code	Code/Data	✓	✓	✓	✓	✓	`lea RBX, [RIP + ℓ]`
S7		(**F5**) `RIP` + Label ± Const.	Code	Code/Data	✗	✗	✗	✓	✓	`lea RAX, [RIP + ℓ + 42]`

† ① is employed by RetroWrite [17], Egalito [52], and Verbeek *et al.* [46]. ‡ ② is employed by Uroboros [48] and Egalito [52].

2.4 Symbolic Labels in Assembly

Symbolization is the process of recovering symbolic labels from numeric values, which is the core step in reassembly-based binary rewriting. To understand the challenges of symbolization, we first categorize what kind of labels compilers can generate, and how they are represented in assembly. Particularly, we divide symbolic labels into seven groups based on (1) what they represent, (2) their corresponding assembly expression format, (3) their location, and (4) the type of pointer they are evaluated to. Table 1 summarizes the seven categories of symbolic labels that compilers can generate for x86-64 binaries, which are denoted as **S1** to **S7** in the first column. Our categorization is based on the previous work by Kim *et al.* [26], but we further refine it by considering expression formats and pointer types. Specifically, Type I, Type II, and Type IV in [26] correspond to **S1**, **S2**, and **S7**, respectively. Type III is divided into **S5** and **S6**, and Type VII is divided into **S3** and **S4**. We exclude Type V and Type VI, as they are not relevant to x64 binaries.

2.4.1 Representation. Compilers generate symbolic labels to represent two kinds of values: absolute addresses and relative offsets. The second column of Table 1 makes a distinction between them. Absolute addresses are a plain pointer to a memory location. They typically point to a global variable or a function. Relative offsets are a relative distance from a base address. They are only meaningful when they are added to a base address. Note that we group **S3** and **S4** in the absolute address category as each label represents an absolute address, although the aggregated expression means a relative offset.

2.4.2 Assembly Expression Format. In x86-64 assembly, symbolic labels can appear in five different expression formats, denoted as **F1**–**F5** in the third column of Table 1. **F1** and **F4** are a plain label that represents either an absolute address or a relative offset. The rest (**F2**, **F3**, and **F5**) are composite expressions that are composed of a label and other expression(s). Composite expressions are the main source of symbolization errors, as Kim *et al.* [26] showed. The last column of the table presents an example assembly for each format.

2.4.3 Location of Symbolic Labels. Symbolic labels can appear either in code or data sections. The third column of Table 1 denotes in which section each kind of symbolic label can appear. For brevity, we merge some of the categories by denoting them as "Code/Data" in the table as they share the same characteristics. For example, **S1** represents a plain label located in either a code or data section.

S4 can only be generated by compilers because high-level language standards (i.e., C, C++, and Fortran) do not define the behavior of adding or subtracting code pointers. We find that **S4** can only appear in a data section to represent an array of compiler-generated offsets, such as jump tables, for x86-64 binaries. **S5** can only appear through a branch instruction, e.g., `jmp Label`, hence, it should be in a code section.

```
; .section .text
0x01414b9: lea RAX, [RIP+0x64b08] # 0x1a5fc8
...
; .section .data.rel.ro.local
0x01a5e40: 0766 0300 0000 0000    # var
...
; .section .fini_array
0x01a5fc8: 0635 1100 0000 0000    # var+392
```

Figure 1. Example taken from `addr2line`, Binutils.

2.4.4 Pointer Type. Symbolic labels, regardless of whether they represent absolute addresses or relative offsets, are eventually evaluated to a pointer of code or data. The fifth column of Table 1 presents what is the ultimate target of each kind of symbolic expressions when they are evaluated to a pointer. For example, consider an x86-64 instruction "`lea RAX, [RIP + 0x41]`" that loads a function pointer into the `RAX` register. As the memory expression "`[RIP + 0x41]`" will be evaluated to a *code* pointer, its pointer type is "Code".

2.5 Symbolization Challenges and Solutions

Each kind of symbolic labels (**S1**–**S7**) has its own challenges in terms of symbolization. Previous works [17, 46, 48, 52] have proposed two major solutions (① and ②) to address the symbolization challenges. The "Impact of Solutions" columns of Table 1 present how each solution addresses the challenges. The "Without Solution" column shows which symbolic labels need no special treatment of existing solutions. Indeed, relative branch targets and plain RIP-relative addresses are readily available by disassembling the binary (**S5** and **S6**). We include them in the table only for the sake of completeness.

2.5.1 Focusing on PIEs (Solution ①). RetroWrite [17], Egalito [52], as well as Verbeek *et al.* [46], shift their focus to PIE binaries to ease the symbolization problem. Since absolute addresses in PIE binaries need to be relocated at load time, compilers always generate relocation information for them. This means one can easily distinguish between addresses and numerals by checking the existence of relocation information; thus, handling **S1** becomes trivial. Nevertheless, correctly recovering composite symbolic expressions is challenging even for PIE binaries, as the seventh column of Table 1 indicates.

2.5.2 Fixing the Layouts of Data Sections (Solution ②). Handling composite symbolic expressions, which involve arithmetic operations on symbolic labels, is the biggest hurdle in symbolization. Unfortunately, compilers often generate composite expressions to optimize data access. For example, a composite expression of the form "Label + 0x42" enables direct access without having to load the base address into a register and add the offset to it.

However, composite expressions can represent a temporary (and potentially invalid) pointer that is later used to form a final (and valid) pointer. Such a temporary pointer may not point to a valid memory location because the compiler generates it temporarily, just to be able to *dynamically* compute the final pointer. For example, it may point to another section, in which case symbolizing the temporary pointer can lead to a runtime error.

```
; .section .text
0x0001320: endbr64
...
0x00064f1: lea  RDI, [RIP+0xfffffffffffe918]# 0x01320
...
0x000c9dd: lea  R9,  [RIP+0x4a74]           # 0x11458
...
0x000c9ea: subsd XMM0, QWORD PTR [R9+RCX*1]
...
0x0011458: mov  EAX, DWORD PTR [RSP+0x4c]   # var-142600
...
; .section .bss
0x0034160: 0000 0000 0000 0000              # var
```

Figure 2. Example taken from `434.zeusmp`, a SPEC CPU2006 binary, where a temporary pointer is pointing to an instruction in the middle of a function.

Figure 1 illustrates such a case. The memory operand of the `lea` instruction points to an element of the `.fini_array` section, but the original symbolic expression from the compiler-generated assembly code is `[RIP + var + 392]`, where `var` is a variable defined in the `.data.rel.ro.local` section. If a reassembler puts a label ℓ at the address `0x1a5fc8` and symbolizes the operand to be `[RIP +` ℓ`]`, then the final pointer derived from this expression will become invalid depending on the relative distance between the two sections.

To mitigate this problem, Uroboros [48] and Egalito [52] lock the layouts of data sections in the rewritten binary, assuming that data modification is not allowed. With this solution, we can guarantee the validity of the final pointer even though reassemblers incorrectly symbolize the temporary pointer to `[RIP +` ℓ`]` because relative distances between data sections are always preserved. However, this solution fails when a data pointer points to a code section or vice versa. Furthermore, we cannot simply fix the layout of the code section as it would prevent instrumenting the code, which suggests the need for solution ③ (§2.6.1).

2.6 Remaining Challenges and Our Solutions

The eighth column of Table 1 shows that the existing solutions (① and ②) can only address a subset of the challenges. There are three types (**S2**, **S4**, and **S7**) that have no sound solution yet. Thus, we describe what are the remaining challenges, and then motivate our solutions (③ and ④).

2.6.1 CET-based Pointer Repairing (Solution ③). Recall from §2.5.2, locking the layouts of data sections is insufficient to handle cases where a temporary code pointer is pointing to a data section, or a temporary data pointer is pointing to a code section. Figure 2 illustrates such a case, where `R9` stores a temporary pointer, which will be eventually used to point to a global variable in the `.bss` section.

```
; .section .text
0x012a71b: lea      RDX, [RIP+0xc885e] # 0x1f2f80
0x012a722: movsxd   RCX, DWORD PTR [RDX+RCX*4]
0x012a726: add      RCX, RDX
0x012a729: notrack jmp RCX
...
; .section .rodata
0x01f2f80: ac77 f3ff                # start of jump table
...
0x01f2fd0: a378 f3ff                # end of jump table
0x01f2fd4: 25e7 fbff                # var
```

Figure 3. Example taken from 600.perlbench_s, SPEC CPU2017, showing the jump table symbolization challenge.

However, the temporary pointer is pointing to a valid instruction at 0x11458 in the .text section, even though it is not a function entry point. If we assign a label to 0x11458 and symbolize the operand of the lea instruction at 0xc9dd, then R9 can have a wrong temporary value when the distance between the two instructions changes via instrumentation.

To address the problem, we propose a novel solution that leverages the characteristics of CET-enabled binaries. In particular, we *statically* identify pointers that are evaluated to a code pointer by checking the presence of CET-specific instructions (i.e., endbr64) inserted by compilers (see §3.4).

2.6.2 Superset Symbolization (Solution ④).

Assembly expression of the form "Label - Label" (**F3**) represents a compiler-generated offset. There are two kinds (**S3** and **S4**), but **S3** is not our interest because such expressions are used to represent a data pointer, which can be handled by ②. According to our study, **S4** can only represent a jump table entry, where each entry represents a relative offset from a base address to a target address. Unfortunately, we find that recovering such symbolic labels is as difficult as recovering a complete CFG of the binary.

More specifically, predicting the boundary of the jump table is challenging because compilers often do not emit bounds-checking logic when the table indices are statically determined [33]. Existing reassemblers [17, 22, 46, 52] employ several heuristics to recover jump tables, but they often incorrectly predict the boundaries of jump tables, leading to symbolization errors.

Figure 3 demonstrates why symbolizing jump table entries is challenging. The RDX register stores the base address (0x1f2f80), which is the start address of a jump table in the .rodata section. The jump table contains 21 entries, each of which is a four-byte offset from the base address to a target address. However, binary analysis tools can easily misidentify the boundary of the table because the global array (located at 0x1f2fd4) immediately follows the last entry of the jump table. Furthermore, the global array contains seemingly valid offsets, which can form a valid pointer when they are added to the base address.

We handle this problem by considering all possible jump table entries, thereby constructing *superset* CFGs that contain

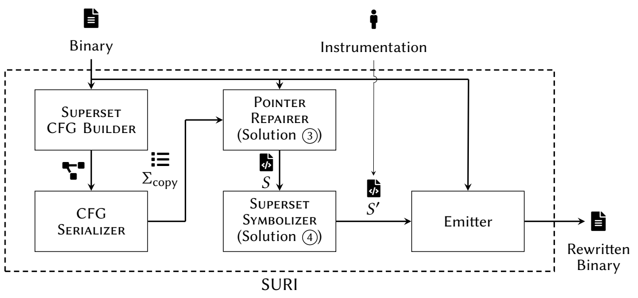


Figure 4. SURI architecture.

all possible indirect branch targets. A superset CFG may contain invalid nodes, and thus, the rewritten binary with the CFG can include unreachable code, but it does *not* affect the correctness of the rewritten binary because such code will never be executed. Such an over-approximation, however, introduces another challenge as it can disturb the correctness of the jump table recovery algorithm itself. Thus, we propose a novel way to soundly symbolize jump table entries with superset CFGs (see §3.5).

3 Design

In this section, we first introduce the overall architecture of SURI and then describe the challenges involved in designing SURI and how we address them.

3.1 SURI Architecture Overview

At a high level, SURI takes in a binary as input and returns a rewritten binary as output. Specifically, SURI runs in five major steps as illustrated in Figure 4.

1. Superset CFG Builder takes in a stripped binary as input, and constructs superset CFGs by recursively disassembling the binary (§3.2).
2. CFG Serializer takes in superset CFGs as input, and transforms the CFGs into a sequence of assembly instructions Σ_{copy}, which constitutes the executable code section of the rewritten binary (§3.3).
3. Pointer Repairer takes in Σ_{copy} and the original binary as input, and returns an intermediate assembly file S, which contains the sections in the original binary as well as a new section for Σ_{copy}. It fixes up every cross-reference in Σ_{copy} to point to the original code/data sections except for the case where the cross-reference evaluates to a valid branch target by leveraging endbr64 instructions (§3.4).
4. Superset Symbolizer takes in an intermediate assembly file S as input, modifies S to soundly symbolize jump tables, and returns a modified assembly file S'. Users can modify S' at this stage to add instrumentation (e.g., adding a security monitor) to it (§3.5).

5. EMITTER converts S' into a binary while ensuring that the original code and data sections are located at the same addresses as in the original binary (§3.6).

Note that the main novelty of SURI lies in the use of superset CFGs to address the symbolization challenges in reassembly, but not in the use of superset CFGs themselves. As we will discuss in the rest of this section, using superset CFGs introduces additional challenges in terms of handling bogus nodes and edges. In order to over-approximate jump tables, we need to repeatedly perform dataflow analysis whenever we encounter a new indirect edge in the superset CFGs (§3.2). We also need to emit overlapping basic blocks in a way that does not affect the semantics of the rewritten binary (§3.3). Furthermore, we need to handle over-approximated jump tables in order to soundly execute the rewritten binary (§3.5).

Another key novelty of SURI is that it identifies pointers that evaluate to a code pointer by leveraging `endbr64` instructions in order to reliably symbolize code pointers in the rewritten binary (§3.4).

3.2 Superset CFG Builder

Existing recursive disassemblers often fail to cover critical code blocks due to their unsound heuristics for recovering indirect branch targets. SOTA reassemblers, such as Egalito [52] and Ddisasm [22], also suffer from the same problem.

To address this, we over-approximate possible (and reachable) indirect edges to construct *superset CFG*s, which are CFGs that include every node and every edge in the original CFGs while potentially including bogus nodes and edges. Those bogus nodes and edges, however, will *not* affect the execution of the rewritten binary as they will never be executed. To our knowledge, SURI is the first to construct superset CFGs to handle symbolization challenges in reassembly.

There are two main steps in recovering superset CFGs. First, SURI collects an initial set of function entry points from a given binary (§3.2.1). Second, it recursively disassembles the binary from each entry point to construct superset CFGs while over-approximating indirect branch targets (§3.2.2).

3.2.1 Harvesting Entry Points. SURI collects a set of determinate function entry points to start our analysis. Specifically, it first identifies the program entry point (i.e., `_start`) from the ELF header, and collects all function pointers from the relocation section. Then, it recursively disassembles each of those function entry points to build superset CFGs for them in the next step.

While it is sufficient to generate superset CFGs with them, we can optimize the CFG recovery process by gathering more function entry points. With more entry points, we can have tighter function boundaries, which can help reduce the number of bogus nodes and edges in our superset CFGs.

To harvest more function entry points, SURI leverages three *conservative* heuristics. First, we consider direct branch targets that lie outside the current function boundary as additional function entry points. Second, we include RIP-relative addresses as a function entry point if they point to an `endbr64` instruction. Finally, we leverage call frame information stored in the `.eh_frame` section, which includes critical data for stack unwinding [38], to recover function entry points. Note we do *not* rely on call frame information to operate, as it merely helps improve the efficiency of the superset CFG construction. All these heuristics are conservative in that they do not affect the correctness of the rewritten binary, but can help reduce the number of bogus nodes and edges in our superset CFGs.

Furthermore, our approach ensures that all reachable code regions are included in the resulting superset CFGs. If there is a function that is not referenced by either a direct branch or a code pointer, it can be considered dead code, and thus, it does not affect the reliability of the rewritten binary.

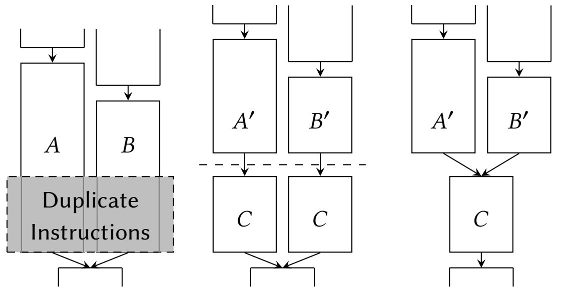
3.2.2 Building Superset CFGs. For each function entry point, SURI first creates an intermediate CFG by recursively following every direct branch target. After recovering all reachable direct edges, it then statically analyzes jump table entries and recursively follows them to construct superset CFGs. A unique design choice of SURI is that it always over-approximates jump tables as well as their entries.

Particularly, it performs backward slicing from each indirect branch instruction (e.g., `jmp rdx`) to recover a symbolic expression for the target register of the form "`base_address + index * 4`". It then calculates all possible values of `base_address` using a classic dataflow analysis. Unlike existing approaches [12], however, SURI always performs data flow analysis whenever a new indirect edge is encountered in order to ensure that there is no missing jump table.

Typically, there should only be a single base address, but our over-approximated analysis can include bogus data flows, which can potentially result in multiple addresses, as we will discuss in §3.5. SURI decides the range of each jump table by checking whether each jump table entry evaluates to a valid code address in the current function boundary, which is the range between the current function entry point and the next function entry point.

Additionally, SURI includes fall-through edges after call instructions without relying on non-returning function analysis as long as the edges remain within the current function boundary. This approach aids in incorporating indirect edges into the CFG, thereby ensuring its completeness.

Since SURI over-approximates indirect branch targets, our superset CFGs can include overlapping basic blocks. In such cases, we merge those blocks to remove duplicates. Figure 5 describes the merging process. First, SURI detects duplicate basic blocks A and B during the CFG construction (Figure 5a). Second, SURI splits the block A to A' and C, and the block B to B' and C (Figure 5b). Finally, SURI creates a merged block C and connects A' to C and B' to C (Figure 5c). The trimmed



(a) Discover. **(b)** Split. **(c)** Merge.

Figure 5. Merging overlapping basic blocks.

basic blocks need special treatment during serialization, as discussed in §3.3.

3.3 CFG Serializer

After constructing superset CFGs, we transform them into a sequence of assembly instructions, which we call Σ_{copy} in this paper. To handle implicit control flows introduced by overlapping basic blocks, we explicitly add a branch instruction to non-overlapping basic block(s) to ensure that the control flows can merge into the overlapping basic block when the emitted assembly code runs.

Algorithm 1 shows the serialization algorithm handling overlapping basic blocks. First, we sort all the basic blocks in G, a superset CFG, by their addresses and store them in list B (Line 3). We then iterate the sorted list B until it becomes empty. At each iteration, we first pick the basic block b with the lowest address (Line 5). We then obtain a set of basic blocks overlapped with b and store them in O (Line 6). If there is no overlapping basic block, we simply disassemble b and append the instructions to the output list *code* (Line 19). If otherwise, we iterate over the overlapped basic blocks O in a nested loop to make control flows explicit by selectively modifying them (Line 10). We first get an overlapping basic block o from O, which has the lowest address (Line 11). We then check if o has a fall-through basic block f (Line 13). If f exists and there exists another overlapping basic block in O, then we add a direct branch instruction to o (Line 15). Note we do not add a branch instruction to the last overlapping basic block, as it will be directly connected to f. Finally, we accumulate instructions by disassembling o and append them to the output list *code* (Line 16).

3.4 Pointer Repairer

Recall from §2.6.1 that we aim to reliably symbolize composite expressions found in **S2** and **S7** by preserving the original sections while making a copy of the original code section. However, we do not rely on the dynamic resolution of every indirect branch target as in Multiverse [5] since it incurs a significant runtime overhead. Instead, we statically figure out which pointers in the target binary will eventually

Algorithm 1: CFG Serialization.

```
1  function cfg_serialization($G$)
2      $code \leftarrow [\cdot]$
3      $B \leftarrow G$.BBLs
4      while $B \neq \emptyset$ do
5          $b \leftarrow$ get_bbl_of_lowest_addr($B$)
6          $O \leftarrow$ get_overlapped_bbls($G$, $b$)
7          if $O \neq \emptyset$ then
8              $O \leftarrow O - \{b\}$
9              $B \leftarrow B - O$
10             while $O \neq \emptyset$ do
11                 $o \leftarrow$ get_bbl_of_lowest_addr($O$)
12                 $O \leftarrow O - \{o\}$
13                 $f \leftarrow$ get_fallthrough_bbl($G$, $o$)
14                 if $f \neq \emptyset$ and $O \neq \emptyset$ then
15                     $o \leftarrow$ add_br_instruction($o$, $f$)
16                 $code \leftarrow$ append_code($code$, $o$)
17         else
18             $B \leftarrow B - \{b\}$
19             $code \leftarrow$ append_code($code$, $b$)
20     return $code$
```

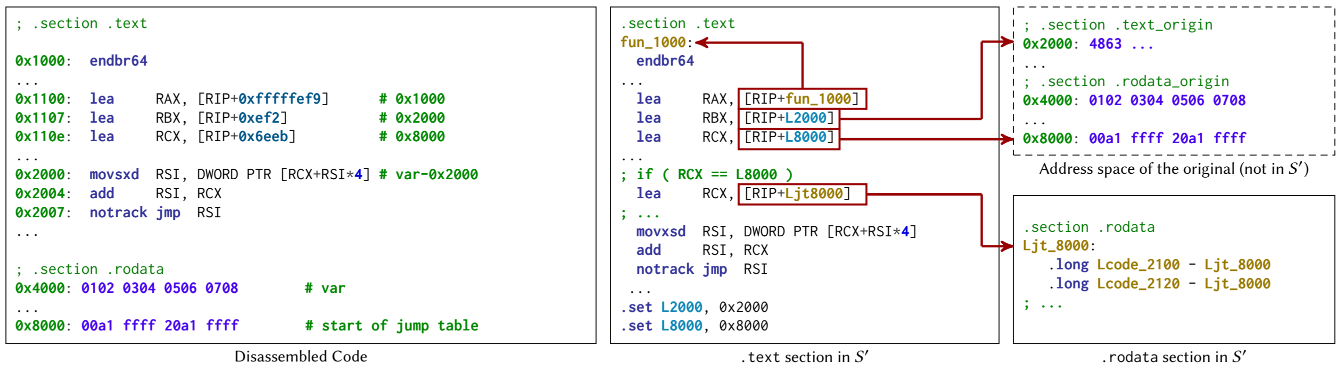
point to a branch target using `endbr64` instructions, which are designed to mark valid destinations for indirect jumps. We then make every code/data pointer in the copied code section point to the original code/data section, except for those pointers that reference an `endbr64` instruction.

Specifically, SURI takes the following two steps. First, it creates a new assembly file S that includes the serialized assembly instructions Σ_{copy}. Second, it identifies all the pointers used in S, symbolizes them depending on their target, and returns S as output. If the target is an `endbr64` instruction, it puts a label on the instruction and makes the pointer to reference the label. Otherwise, it makes the pointer to point to the original code/data sections. This way, we can safely instrument the copied code while preserving the values in temporary pointers.

To preserve the original sections, we use the `.set` directive of GNU AS to assign a particular address to a label. Figure 6 shows an example usage of the `.set` directive. The expression "`.set .L8000, 0x8000`" defines the label `.L8000` that references the address `0x8000`. Although the assembly file does not define any code/data in the address, the `.set` directive allows us to define a label referencing a non-existing address. The final binary will eventually include the code/data in the address `0x8000` as EMITTER will keep the original sections at the same addresses as in the original binary (§3.6).

3.5 Superset Symbolizer

After repairing all the pointers in S, SURI creates new jump tables and adds them to S to produce S'. Recall from §2.6.2 that we address the jump table symbolization challenge by over-approximating jump table entries with superset CFGs.



```
; .section .text

0x1000:  endbr64
...
0x1100:  lea     RAX, [RIP+0xfffffef9]      # 0x1000
0x1107:  lea     RBX, [RIP+0xef2]           # 0x2000
0x110e:  lea     RCX, [RIP+0x6eeb]          # 0x8000
...
0x2000:  movsxd  RSI, DWORD PTR [RCX+RSI*4] # var-0x2000
0x2004:  add     RSI, RCX
0x2007:  notrack jmp  RSI
...

; .section .rodata
0x4000: 0102 0304 0506 0708        # var
...
0x8000: 00a1 ffff 20a1 ffff        # start of jump table
```

Disassembled Code

```
.section .text
fun_1000:
  endbr64
...
  lea     RAX, [RIP+fun_1000]
  lea     RBX, [RIP+L2000]
  lea     RCX, [RIP+L8000]
...
; if ( RCX == L8000 )
  lea     RCX, [RIP+Ljt8000]
; ...
  movxsd  RSI, DWORD PTR [RCX+RSI*4]
  add     RSI, RCX
  notrack jmp  RSI
 ...
.set L2000, 0x2000
.set L8000, 0x8000
```

.text section in S'

```
; .section .text_origin
0x2000: 4863 ...
...
; .section .rodata_origin
0x4000: 0102 0304 0506 0708
...
0x8000: 00a1 ffff 20a1 ffff
```

Address space of the original (not in S')

```
.section .rodata
Ljt_8000:
   .long Lcode_2100 - Ljt_8000
   .long Lcode_2120 - Ljt_8000
; ...
```

.rodata section in S'

Figure 6. Example of supserset symbolization.

However, we face two additional problems when symbolizing jump tables with superset CFGs. First, by over-approximating jump table entries, we may *falsely* symbolize irrelevant data values, which will corrupt the original data. We address this problem by isolating the over-approximated jump tables from the original data section (§3.5.1). Second, our superset CFG includes bogus nodes and edges, which can introduce bogus data flows between registers and memory locations, thereby affecting the correctness of our static analysis. In particular, we may incorrectly approximate the base address of a jump table, and thus obtain two or more base addresses for a single jump table. We address this problem by dynamically identifying the right base address (§3.5.2).

3.5.1 Jump Table Isolation. Our over-approximated jump tables can naturally include false entries. For example, we may include other global variables (e.g., `var` in Figure 3) as jump table entries. To address this problem, we allocate a new read-only data section to exclusively store the over-approximated jump tables. Figure 6 illustrates how SURI symbolizes jump tables. The bottom right of the figure shows the newly created jump table (`Ljt_8000`), which includes over-approximated jump offsets. `Lcode_2100` and `Lcode_2120` are indirect branch targets, and each entry stores the difference between the target address and the base address of the jump table (`Ljt_8000`). Since we separately store each of the over-approximated jump tables in a newly allocated `.rodata` section, we can safely store overlapping jump tables, which may include bogus entries, without corrupting the original data.

3.5.2 Dynamic Base Identification. As superset CFGs can include bogus nodes and edges, we may incorrectly approximate the base address of a jump table during the static analysis. Note, however, our approximation is conservative, meaning that our analysis will not miss the correct base address, although it may include wrong addresses. We tackle this problem by adding instrumentation code to dynamically fix up the base address.

For example, the third `lea` instruction in Figure 6 loads the address of the jump table to `RCX`, and uses it to compute a jump target with the `movxsd` instruction. The problem happens if our static analysis over-approximates the value of `RCX` to be either `0x8000` and `0x9000`, one of which is correct and the other is wrong. In such a case, we cannot statically determine which is the right one. To handle this problem, we insert if-then-else statements (in assembly) to dynamically check the base address at runtime, and assign the right address, which is located at the newly allocated `.rodata` section, to `RCX` (as shown in the fourth `lea` instruction in the middle of Figure 6).

Although our dynamic base identification approach can incur a runtime overhead, it is negligible in practice as our experiments show (§4.3.1). This is mainly because we do not need to add if-then-else statements for every indirect jump, but only for those that our static analysis incorrectly identifies two or more base addresses.

3.6 Emitter

Emitter takes in as input the original binary as well as the instrumented assembly file S', and outputs a rewritten binary. At a high level, it first compiles S' into a binary file, and then appends the sections in the binary to the original binary while preserving the layouts of the original sections.

Specifically, we first create a binary from S' using the linker option `--section-start` in order to preserve the relative distances between the newly created sections and the original sections. This way, we can also avoid any overlap between the original sections and the newly added sections. We then extract all the text and data sections from the newly created binary, and append them to the original binary. Figure 7 describes this process.

Note that appending sections to a binary is not a trivial task, as we need to update various metadata in the ELF file. At a high level, SURI performs the following steps before emitting a final binary.

- Update the program header and section table to include newly added segments and sections.

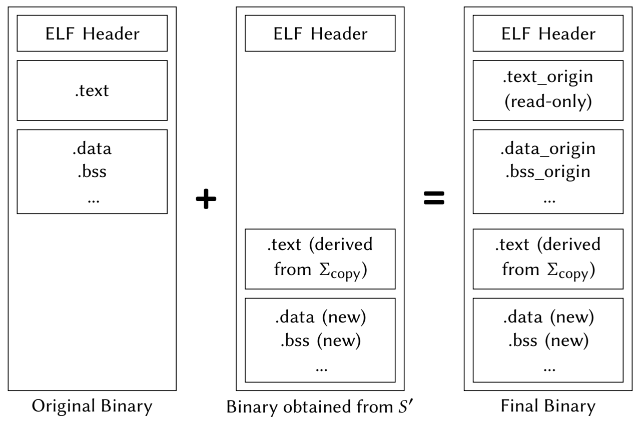


Figure 7. Layout Preservation

- Change the access permission of the original code section to read-only.
- Merge symbol tables and string tables of both binaries.
- Adjust relocation entries in the original binary to reference the newly added code addresses.
- Update the dynamic section to correctly use updated symbol, string, and relocation tables.

3.7 Implementation

We implemented SURI with approximately 4.5K SLoC of Python and 1.1K SLoC of F#. To build superset CFGs, we modified B2R2 [25], a binary analysis platform that supports x86-64 binary lifting. Our jump table analysis is implemented on top of B2R2's Intermediate Representation (IR). Additionally, we used pyelftools [7] to parse exception handling information from the binary.

4 Evaluation

In this section, we evaluate SURI to answer the following research questions.

RQ1. How well does SURI compare to the state-of-the-art reassembly tools in terms of reliability? (§4.2)

RQ2. How big is the performance overhead introduced by SURI for rewritten binaries? (§4.3)

RQ3. Is SURI applicable to real-world scenarios, such as runtime memory sanitization? (§4.4)

4.1 Experiment Setup

4.1.1 Benchmark. To measure the reliability of reassemblers, one needs to run rewritten binaries with a large number of test cases to exercise diverse execution paths. Thus, we chose four popular software packages that include well-maintained test suites: Coreutils v9.1 (108 programs), Binutils v2.40 (15 programs), SPEC CPU2006 (31 programs), and SPEC CPU2017 (47 programs), which are written in C, C++, and Fortran. The SPEC benchmarks include complex real-world programs such as GCC, GNU assembler, GNU linker, Perl, bzip2, and x264 encoder/decoder. Also, we used four major compilers (GCC v11.0, GCC v13.0, Clang v10.0, and Clang v13.0) and two linkers (GNU ld v2.34 and GNU gold v1.16) to produce our dataset with varying optimization levels (`O0`, `O1`, `O2`, `O3`, `Os`, and `Ofast`) to obtain a diverse set of binaries.

This gives us 48 (= $4 \times 2 \times 6$) different configurations per binary, and, hence, a total of 9,648 (= 48×201) CET-enabled PIE binaries. Out of those 9,648 binaries, we had to exclude 4, 42, and 2 binaries from Coreutils v9.1, SPEC CPU2006, and SPEC CPU2017, respectively, because they simply did not pass the test suites. Therefore, our final benchmark consists of 9,600 binaries. All the binaries we used in our experiments are *stripped*, i.e., they do not contain any debugging information. To our knowledge, this is the *largest dataset* used in reassembly research to date.

4.1.2 Measuring Reliability. Rewritten binaries may function correctly in some cases but fail in others because symbolization errors would only affect a subset of execution paths. Therefore, we measured the reliability of each reassembler by running the test suites provided by each software package on the rewritten binaries. One may consider using automated test case generation tools to have a more comprehensive test suite, but we leave it as future work since manually written test suites are already sufficient to demonstrate the effectiveness of our approach: (1) the test suites are well maintained and achieve a high code coverage, and (2) the test suites can already expose many symbolization errors of the SOTA reassemblers we compared against.

Since Coreutils and Binutils do not provide a separate test suite for each individual program, we counted the entire test suite as a single test. That is, if one of the Coreutils/Binutils binaries rewritten by a tool failed to pass the test suite, we consider the tool to have failed the entire test. For SPEC benchmarks, we counted each individual test as a separate test. This choice may underestimate the symbolization errors for Coreutils and Binutils, but it is sufficient to demonstrate the effectiveness of our approach since it only makes the comparison more favorable to other reassemblers—SURI has no symbolization errors in our experiments.

4.1.3 Comparison Target. We selected two state-of-the-art tools for comparison: Ddisasm [22] v1.7.0 (docker image digests `7b6c27`, Sep. 2023), and Egalito [52] (commit `c5bccb`, Jun. 2020). We excluded RetroWrite [17] since it cannot reassemble stripped binaries. We also excluded the tool presented by Verbeek *et al.* [46] as it requires user interaction to resolve function pointers. Furthermore, we excluded patch-based rewriting tools, such as E9Patch [18], and table-driven rewriting tools, such as Multiverse [5], as they are fundamentally different from SURI in that they do not perform symbolization. We further discuss the limitations of the other approaches in §6.

Table 2. Comparison against Ddisasm in terms of rewriting completion rate, rewriting time, and test suite pass rate.

		SURI			Ddisasm		
		Fin.[†]	T (s)[‡]	Pass	Fin.	T (s)	Pass
Coreutils	Clang	100%	6.4	Succ*	100%	1.6	Succ*
	GCC	100%	6.3	Succ*	100%	1.5	Fail*
Binutils	Clang	100%	31.5	Succ*	100%	51.7	Succ*
	GCC	100%	31.8	Succ*	100%	47.6	Fail*
SPEC 2006	Clang	100%	27.8	100%	100%	67.3	86.9%
	GCC	100%	27.9	100%	100%	71.9	85.9%
SPEC 2017	Clang	100%	97.7	100%	95.6%	194.7	84.7%
	GCC	100%	102.8	100%	99.1%	208.1	82.1%

[†] Rate of successfully rewriting binaries without errors.
[‡] Time taken (in seconds) to run the tool.
* Coreutils and Binutils do not support per-program testing. Thus, we report either success (Succ) or failure (Fail). See §4.1.2 for more details.

4.2 Reliability Comparison

To answer **RQ1**, we measured the reliability of three reassemblers, including SURI, Ddisasm, and Egalito, following the methodology described in §4.1.2, and compared them against each other. We used two different operating systems, as Egalito could not run on binaries compiled with Ubuntu 20.04 but worked with those from Ubuntu 18.04. Also, we used a reduced set of binaries for comparing Egalito because it does not support C++ binaries, and Ubuntu 18.04 is limited to GCC v11.0 and Clang v10.0. When comparing rewriting time, we only considered binaries that were successfully rewritten by all the tools of interest.

4.2.1 Comparison Against Ddisasm. Table 2 presents the comparison against Ddisasm, which individually shows the results for GCC- and Clang-compiled binaries for each package. First of all, SURI was the only tool that successfully rewrote all the binaries in our benchmark. Although Ddisasm was able to rewrite most of them, it failed to rewrite 8.5% of the programs in SPEC CPU2017 as it produced invalid labels in the resulting assembly code, making the compilers fail to compile it. In terms of rewriting time, both tools were similar, but SURI was slightly faster than Ddisasm. However, the difference was just a matter of a few minutes at most, and furthermore, rewriting speed is not a critical factor in practice as it is a one-time cost.

Moreover, SURI successfully passed all the tests, whereas Ddisasm faced challenges in passing all test suites for Coreutils and Bintuils and failed in more than 16.1% of the tests in SPEC benchmarks. Note that the "Pass" columns present the pass rates of each tool for those binaries that were successfully rewritten, i.e., the pass rates of each tool for the entire benchmark are lower than the ones presented in the table if we consider the failures in rewriting. This result clearly demonstrates the superior reliability of our rewriting technique compared to Ddisasm, the SOTA reassembler.

Table 3. Comparison against Egalito in terms of rewriting completion rate, rewriting time, and test suite pass rate. We omitted C++ binaries here as Egalito does not support them.

		SURI			Egalito		
		Fin.	T (s)	Pass	Fin.	T (s)	Pass
Coreutils	Clang	100%	7.3	Succ	100%	0.1	Fail
	GCC	100%	6.1	Succ	89.7%	0.1	Fail
Binutils	Clang	100%	37.0	Succ	96.7%	2.2	Fail
	GCC	100%	33.0	Succ	87.8%	2.0	Fail
SPEC 2006	Clang	100%	27.1	100%	98.9%	1.4	92.1%
	GCC	100%	26.8	100%	93.2%	1.4	79.9%
SPEC 2017	Clang	100%	96.2	100%	97.4%	5.5	85.7%
	GCC	100%	115.1	100%	90.9%	7.1	73.2%

4.2.2 Comparison Against Egalito. Table 3 presents the comparison against Egalito. Egalito was not able to process 5.3% of the binaries in our benchmark due to assertion failures that occurred during the rewriting process. Thus, we excluded those binaries when comparing the rewriting time and test suite pass rate. Egalito was significantly faster than SURI in terms of rewriting speed, but rewriting is a one-time cost, as discussed earlier, and the differences were just a matter of a few minutes. More importantly, SURI clearly outperformed Egalito in terms of reliability. Egalito was unsuccessful in passing all test suites for Coreutils and Binutils, and it failed to pass more than 17.9% of the tests in the SPEC benchmarks, while SURI passed all the tests without any problems.

4.2.3 Reliability of SURI. The experimental results clearly demonstrate the practical impact of SURI; It was the only tool that achieved a 100% rewriting completion rate and a 100% test suite pass rate. However, it is noteworthy that having a 100% pass rate does not necessarily prove the soundness of SURI because there could be uncovered execution paths that can potentially cause symbolization errors. Thus, we do not claim that SURI is sound, but instead, we empirically show its reliability by demonstrating that it can pass all the tests in our benchmark (see §5.1 for further discussion).

To further verify the reliability of SURI, we performed extra experiments with 10 additional programs that are large and complex: Apache-2.4.56, MariaDB-11.5.0, Nginx-1.23.3, SQLite-3.31.2, 7-Zip-24.05, Epiphany-3.36.4, Filezilla-3.46.3, Openssh-8.2p1, Putty-0.73, and Vim-8.1. The first five programs have their own Phoronix test suite [34], while the rest do not. Thus, we manually verified the correctness of the rewritten binaries for the latter five programs. As a result, we were able to confirm that SURI can successfully rewrite all the programs and completely pass all the tests.

4.2.4 Effectiveness of ③ and ④.

To evaluate the effectiveness of our proposed solution ③ and ④, we analyzed the distribution of symbol types in our benchmark binaries. First, we utilize the tool proposed by Kim *et al.* [26] to classify symbols into our symbol types. The resulting distribution of symbols across categories **S1** –**S7** is as follows: 5.9%, 0.2%, 0.1%, 4.6%, 67.7%, 18.9%, and 2.7%. Therefore, symbols categorized under **S2** and **S7**, which are addressed by solution ③, account for 2.9% of the total. In contrast, symbols under S4, handled by solution ④, constitute a significant 4.6%.

Second, we examined approximately 8.9 million code pointers identified by SURI and confirmed that every pointer correctly targets valid code. This result demonstrates the effectiveness of using `endbr64` to analyze pointer types. That is, SURI can effectively identify and symbolize code pointers.

Additionally, we note that our experimental results shown earlier demonstrate the effectiveness of our solutions. A single failure in symbolization can lead to runtime errors in the rewritten program, making it a significant challenge to ensure reliability. However, through rigorous testing on a large dataset, SURI achieved a 100% success rate, demonstrating that our proposed methods reliably resolve previously unresolved symbolization challenges.

4.3 Overhead of Rewritten Binaries

How much overhead does SURI introduce for a rewritten binary? We answer this question by measuring the performance difference between the original binaries and the rewritten binaries produced by each tool. To measure the performance, we used a machine with Intel Core i9-11900K equipped with 128GB of RAM. We used a docker container and assigned only a single core to each container to run the tools. To reduce the noise in the measurements, we leveraged only two cores of the machine to run our benchmarks.

4.3.1 Overhead Incurred by SURI.

We rewrote all the SPEC binaries in our benchmark (i.e., both SPEC CPU2006 and SPEC CPU2017) using SURI without any additional instrumentation (i.e., no-op instrumentation), and measured the performance difference between the original binaries and the rewritten binaries. We ran each pair of binaries three times and took the average of the results. On average, SURI incurred only negligible (0.2%) overhead to the rewritten SPEC binaries.

We further analyzed the instrumentation overhead of SURI by measuring (1) the number of instructions added to the rewritten binaries; (2) the number of if-then-else statements inserted for undecided jump table addresses; and (3) the number of over-approximated jump table entries. On average, SURI added only 2.8% more instructions to the rewritten binaries, inserted if-then-else statements for about 1.9% of jump tables, and added approximately 9.7% more jump table entries due to over-approximation.

Table 4. Runtime overhead of rewritten SPEC binaries produced by SURI, Ddisasm, and Egalito. We used the binaries compiled with the `O3` optimization level.

	# of Bins	Ubuntu18.04		Ubuntu20.04	
		SURI	Egalito	SURI	Ddisasm
SPEC CPU2006	24	0.46%	0.69%	0.33%	0.32%
SPEC CPU2017	21	0.17%	0.04%	0.12%	0.31%
Total	45	0.32%	0.39%	0.26%	0.32%

4.3.2 Comparison against SOTA Reassemblers.

To understand how SURI compares to the other reassemblers (Ddisasm and Egalito), we first collected SPEC binaries that were compiled with the `O3` optimization level, and then filtered out binaries that were not successfully rewritten by all three reassemblers. We used no-op instrumentation to measure the overhead introduced by each tool.

Table 4 summarizes the averaged overhead results over three runs for each tool. We can see that all three tools incurred negligible overhead to the rewritten binaries. This result demonstrates that the performance of SURI is comparable to the state-of-the-art reassemblers while it can reliably rewrite all the binaries in our benchmark at the same time.

4.3.3 Impact of Call Frame Information.

We conducted additional experiments to evaluate the impact of using call frame information. To elide call frame information, we produced a dataset with compiler flags `-fno-unwind-tables` and `-fno-asynchronous-unwind-tables`. We then ran SURI on these binaries to evaluate if the absence of call frame information would impact the reliability and performance of the reassembled code.

SURI successfully reassembled all the binaries, regardless of the presence of call frame information. However, SURI required more time for rewriting without call frame information. On average, SURI constructs superset CFGs 4.1× faster with call frame information than without it in our benchmarks. The generated superset CFGs also include 20.2% more instructions when omitting call frame information. Also, we observed that rewritten binaries passed all the test suites even without the presence of call frame information.

Lastly, we measured the runtime overhead of rewritten binaries. We collected 77 SPEC binaries compiled with the O3 optimization level and ran each pair of binaries three times. On average, SURI shows a runtime overhead of approximately 0.23% with call frame information, and 0.65% without it. We believe the additional runtime overhead is due to the increased number of overlapping blocks and if-then-else statements. The result demonstrates that SURI still incurs negligible overhead to rewritten binaries even without call frame information.

Table 5. Memory corruption detection results on Juliet Test Suite binaries.

	Ours	BASan	ASan
True Positives	10,233	9,552	13,378
False Positives	0	8	0
False Negatives	5,528	6,209	2,383
True Negatives	577	569	577
Total Binaries	16,338	16,338	16,338

4.4 Application of SURI

We now demonstrate the applicability of SURI to real-world scenarios by showing that it can be used to instrument binaries for sanitization. In particular, we implemented a binary-only address sanitizer on top of the instrumentation framework of SURI. Our sanitizer adds a runtime check for every memory access to detect out-of-bound memory accesses. Unlike the original address sanitizer (ASan) [43], though, our sanitizer does not sanitize global variables because it is not feasible to identify global variables in a binary without any debugging information. The same limitation applies to BASan, a binary-only address sanitizer implemented on top of RetroWrite [17].

We used the same benchmark (Juliet Test Suite v1.3) used by RetroWrite [17] to evaluate the effectiveness of our binary-only address sanitizer. The authors of RetroWrite reported in their paper that they extracted 11,828 binaries that are related to five CWEs (CWE121, CWE122, CWE124, CWE126, and CWE127) from the Juliet Test Suite. However, we were able to obtain about 5K more binaries (thus, 16,338 binaries in total) that are related to the same CWEs. Therefore, we used the 16,338 binaries extracted from the Juliet Test Suite to evaluate our binary-only address sanitizer.

We first ran our sanitizer as well as BASan (commit `ef4e5`, Nov. 2023) and ASan on the 16,338 programs. Overall, binary-only tools show less precision than source-based ASan due to the inherent limitation of binary-only analysis, e.g., we do not sanitize global variables. However, our sanitizer shows comparable precision to ASan and demonstrates zero false positives. It also shows slightly better precision than BASan, an existing binary-only approach. We further investigated the reason behind the difference in precision between our sanitizer and BASan, and found that it was mainly due to the implementation bug in BASan. Particularly, BASan can corrupt stack memory when saving register values, leading to unintended program behavior.

5 Discussion

5.1 Soundness of SURI

Although our empirical study shows that SURI can soundly reassemble all the binaries in our dataset, it is worth noting that there could be a corner case where our technique may fail to correctly symbolize references. In particular, we may falsely consider a numeral value as an `endbr64` instruction when we analyze pointer types, as we solely rely on the byte pattern of the instruction. As demonstrated in §4.2.4, it is indeed extremely unlikely for a compiler to generate a temporary pointer whose target is `0xf30f1efa`, which is the encoded value of `endbr64`. Nonetheless, our CET-based pointer type detection method could suffer from this particular problem.

For this reason, we do *not* argue that SURI is sound. Instead, this paper systematically finds what are the current requirements for designing a sound reassembler, and suggests a practical method towards sound reassembly.

5.2 Generalizability

Our current focus is on CET-enabled x86-64 binaries, but superset CFG construction and superset symbolization techniques are not dependent on the target ISA. Since SURI leverages an architecture-neutral IR, as discussed in §3.7, it can be readily applied to other architectures. On the other hand, the pointer repairing mechanism (§3.4) currently depends on the Intel architecture, but the idea is generalizable to other architectures, such as ARMv8 where there is a similar hardware-based CFI mechanism, such as BTI [4]. We leave it as future work to extend our technique to ARM binaries. As our technique assumes the existence of a specific hardware feature, it is not generally applicable to legacy binaries. Although we believe that CET is already a popular and general technique for modern Intel CPU products and has been widely adopted by modern Linux OSes, existing reassemblers, such as Ddisasm [22], are still valuable. Furthermore, they can benefit from our technique when dealing with modern binaries.

6 Related Work

Binary rewriting is essential for a wide range of applications as it enables program analysis and instrumentation on binaries. There are techniques that retrofit security monitors into COTS applications [2, 11, 13, 14, 19, 23, 30, 39, 41, 49, 50, 53]. This section discusses the related work on binary rewriting, symbolization, and understanding the characteristics of modern binaries.

6.1 Binary Rewriting

While classical binary rewriting techniques have played a crucial role in patching source-less binaries, they come with limitations concerning the fine-grained binary instrumentation. Patch-based rewriting approaches [8, 15, 18, 24, 29] modify specific code segments while preserving other binary addresses, making them primarily suitable for straightforward binary repairs. Specifically, Detour [24] installs hooks at the start of target functions to intercept function calls. PEBIL [29] and Bistro [15] provide general frameworks to rewrite target functions by inserting branch instructions at the function entries to redirect to a new address. Dyninst [8] offers more comprehensive APIs for inserting branch instructions at target addresses. E9Patch [18] introduces various techniques for installing jump instructions at arbitrary addresses without control flow recovery. However, the approach is limited in the scope of instrumentation points since it overwrites existing code for instrumentation. Additionally, E9Patch may corrupt inlined data in the text section since it does not distinguish between code and data when adding instrumentation. Furthermore, it suffers from high runtime overhead when handling many instrumentation points due to excessive detouring.

There is another class of rewriting approaches that we refer to as table-driven rewriting, where the rewriter maintains mapping tables to resolve indirect branches [5, 42, 51, 56]. In particular, they modify indirect branch instructions to jump to trampoline code, which dynamically computes indirect branch targets. Specifically, REINS [51] and μSBS [42] construct address mapping tables to resolve indirect branches in target binaries. PSI [56] and Multiverse [5] employ two-level address mapping tables to manage indirect branches across shared libraries. Additionally, Multiverse proposes superset disassembly to address the disassembly challenge. Although this approach does support fine-grained instrumentation, it suffers from high performance overhead due to excessive table lookups.

6.2 Symbolization-based Rewriting

Symbolization-based rewriting approaches address the aforementioned limitations. The core idea is to transform a binary into a relocatable form using symbolization techniques, which enables the insertion of fine-grained instrumentation to any point in the code with minimum time and space overhead. To our knowledge, the concept of symbolization is first introduced in Trace-Oriented Programming (TOP) [54], which is a technique to transform an execution trace into relocatable code. Uroboros [48] is the first in adopting the idea of symbolization to statically rewrite binaries. Ramblr [47] proposes systematic approaches to address several symbolization challenges. Ddisasm [22] employs data access patterns to precisely identify code pointers and recover original symbolic expressions. RetroWrite [17] proposes to focus on PIE binaries to circumvent several symbolization challenges. Egalito [52] presents a novel way to this line of research by leveraging metadata of ELF binaries, such as exception handling information. However, they are susceptible to disassembly and symbolization errors.

Recently, there has been an attempt to design a reliable reassembly for AArch64 architecture which has 4-byte wide fixed-length instructions. ARMore [16] disassembles all 4-byte aligned code and installs a rebound table, which redirects control transfer to the corresponding address in reassembled code, into an original code section. However, their solution does not apply to architectures that have variable instruction lengths, such as x86-64. Moreover, ARMore requires call emulation to support C++ binaries, which introduces a significant overhead (about 10% slowdown).

Verbeek *et al.* [46] recently proposes a way to verify the soundness of a reassembled binary, but they do not present a sound reassembly method. Their approach essentially has the same limitations as existing reassemblers as we discussed in §2.5. Their verification method, however, is orthogonal to our work and can be applied to SURI to further validate the soundness of reassembled binaries.

6.3 Analysis of Modern Binaries

Several studies have been conducted to understand and leverage the characteristics of modern binaries. Ghidra [36] and Egalito [52] utilize call frame information, which is now inserted by default by several compilers even for C programs to support C++ interoperability, to disassemble stripped binaries. However, they still encounter disassembly errors due to the imprecision of static analysis. FunSeeker [27] shows that not every compiler generates call frame information and proposes a novel way to identify function entry points by utilizing the characteristics of CET instructions. Additionally, FunProbe [28] recently presents a practical method to identify function entry points even without relying on CET instructions. Currently, our system does not leverage those techniques, but as we showed in §3.2.1, precisely identifying function entry points can benefit SURI in terms of increasing the performance of our superset CFG-based analyses. We leave it as future work to integrate them into SURI.

7 Conclusion

In this paper, we designed and implemented a novel reassembler by identifying and addressing currently unmet needs in symbolization-based reassembly. Particularly, we showed that existing reassemblers suffer from symbolization errors caused by the imprecise identification of dynamically computed pointers and the unsound disassembly of indirect branches. We addressed these issues by (1) leveraging CET instructions to identify whether a pointer is referencing a valid branch target, and (2) using superset CFGs to over-approximate the set of possible branch targets. We implemented our design in a tool called SURI, and showed its efficacy by evaluating it on the largest reassembly benchmark to date, consisting of 9,600 real-world binaries.

Acknowledgements

We thank our shepherd, Sotiris Apostolakis, and the anonymous reviewers for their valuable feedback. This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2021-0-01332, Developing Next-Generation Binary Decompiler).

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 340–353, 2005.

[2] Anil Altinay, Joseph Nash, Taddeus Kroes, Prabhu Rajasekaran, Dixin Zhou, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, and Cristiano Giuffrida. BinRec: Dynamic binary lifting and recompilation. In *Proceedings of the ACM European Conference on Computer Systems*, pages 1–16, 2020.

[3] Arch Linux. Updates to build flags. https://rfc.archlinux.page/0003-buildflags/.

[4] ARM. Branch target identification (BTI). https://developer.arm.com/documentation/ddi0596/2021-06/Base-Instructions/BTI--Branch-Target-Identification-.

[5] Erick Bauman, Zhiqiang Lin, and Kevin Hamlen. Superset disassembly: Statically rewriting x86 binaries without heuristics. In *Proceedings of the Network and Distributed System Security Symposium*, 2018.

[6] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference*, pages 41–46, 2005.

[7] Eli Bendersky. pyelftools. https://github.com/eliben/pyelftools, 2011.

[8] Andrew R Bernat and Barton P Miller. Anywhere, any-time binary instrumentation. In *Proceedings of ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, pages 9–16, 2011.

[9] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 265–275, 2003.

[10] CentOS Git Server. rpms/redhat-rpm-config. https://git.centos.org/rpms/redhat-rpm-config/blob/c8s/f/SOURCES/buildflags.md.

[11] Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive mutational fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 725–741, 2015.

[12] Cristina Cifuentes and Mike Van Emmerik. Recovery of jump table case statements from binary code. In *Proceedings of the International Workshop on Program Comprehension*, pages 192–199, 1999.

[13] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. Ropdefender: A detection tool to defend against return-oriented programming attacks. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security*, pages 40–51, 2011.

[14] Daniele Cono D'Elia, Emilio Coppa, Simone Nicchi, Federico Palmaro, and Lorenzo Cavallaro. SoK: Using dynamic binary instrumentation for security (and how you may get caught red handed). In *Proceedings of the ACM Asia Conference on Computer and Communications Security*, pages 15–27, 2019.

[15] Zhui Deng, Xiangyu Zhang, and Dongyan Xu. BISTRO: Binary component extraction and embedding for software security applications. In *Proceedings of the European Symposium on Research in Computer Security*, pages 200–218, 2013.

[16] Luca Di Bartolomeo, Hossein Moghaddas, and Mathias Payer. ARMore: Pushing love back into binaries. In *Proceedings of the USENIX Security Symposium*, pages 6311–6328, 2023.

[17] S Dinesh, N Burow, D Xu, and M Payer. Retrowrite: Statically instrumenting COTS binaries for fuzzing and sanitization. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 128–142, 2020.

[18] Gregory J Duck, Xiang Gao, and Abhik Roychoudhury. Binary rewriting without control flow recovery. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 151–163, 2020.

[19] Mohamed Elsabagh, Dan Fleck, and Angelos Stavrou. Strict virtual call integrity checking for C++ binaries. In *Proceedings of the ACM Asia Conference on Computer and Communications Security*, pages 140–154, 2017.

[20] Ulfar Erlingsson, Martin Abadi, Michael Vrable, Mihai Budiu, and George C Necula. XFI: Software guards for system address spaces. In *Proceedings of the USENIX Symposium on Operating System Design and Implementation*, pages 75–88, 2006.

[21] Fedora Project Wiki. Changes/hardeningflags28. https://fedoraproject.org/wiki/Changes/HardeningFlags28.

[22] Antonio Flores-Montoya and Eric Schulte. Datalog disassembly. In *Proceedings of the USENIX Security Symposium*, pages 1075–1092, 2020.

[23] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W. Davidson. ILR: Where'd my gadgets go? In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 571–585, 2012.

[24] Galen Hunt and Doug Brubacher. Detours: Binary interception of win32 functions. In *Proceedings of the Conference on USENIX Windows NT Symposium*, 1999.

[25] Minkyu Jung, Soomin Kim, HyungSeok Han, Jaeseung Choi, and Sang Kil Cha. B2R2: Building an efficient front-end for binary analysis. In *Proceedings of the NDSS Workshop on Binary Analysis Research*, 2019.

[26] Hyungseok Kim, Soomin Kim, Junoh Lee, Kangkook Jee, and Sang Kil Cha. Reassembly is hard: A reflection on challenges and strategies. In *Proceedings of the USENIX Security Symposium*, pages 1469–1486, 2023.

[27] Hyungseok Kim, Junoh Lee, Soomin Kim, SeungIl Jung, and Sang Kil Cha. How'd security benefit reverse engineers? the implication of intel CET on function identification. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 559–566, 2022.

[28] Soomin Kim, Hyungseok Kim, and Sang Kil Cha. FunProbe: Probing functions from binary code through probabilistic analysis. In *Proceedings of the International Symposium on Foundations of Software Engineering*, pages 1419–1430, 2023.

[29] Michael Laurenzano, Mustafa M Tikir, Laura Carrington, and Allan Snavely. PEBIL: Efficient static binary instrumentation for linux. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems & Software*, pages 175–183, 2010.

[30] Juanru Li, Zhiqiang Lin, Juan Caballero, Yuanyuan Zhang, and Dawu Gu. K-hunt: Pinpointing insecure cryptographic keys from execution traces. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 412–425, 2018.

[31] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 190–200, 2005.

[32] LWN.net. Kernel release status. https://lwn.net/Articles/924113/.

[33] Xiaozhu Meng and Barton P. Miller. Binary code is not easy. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 24–35, 2016.

[34] Michael Larabel and Matthew Tippett. Phoronix test suite. https://phoronix-test-suite.com.

[35] Susanta Nanda, Wei Li, Lap-Chung Lam, and Tzi-cker Chiueh. BIRD: Binary interpretation using runtime disassembly. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 358–370, 2006.

[36] National Security Agency. Ghidra. https://ghidra-sre.org.

[37] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 89–100, 2007.

[38] Chengbin Pang, Ruotong Yu, Dongpeng Xu, Eric Koskinen, Georgios Portokalidis, and Jun Xu. Towards optimal use of exception handling information for function detection. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 338–349, 2021.

[39] Fei Peng, Zhui Deng, Xiangyu Zhang, Dongyan Xu, Zhiqiang Lin, and Zhendong Su. X-Force: Force-executing binary programs for security applications. In *Proceedings of the USENIX Security Symposium*, pages 829–844, 2014.

[40] Phoronix. Intel shadow stack finally merged for linux 6.6. https://www.phoronix.com/news/Intel-Shadow-Stack-Linux-6.6.

[41] Aravind Prakash, Xunchao Hu, and Heng Yin. vfGuard: Strict protection for virtual function calls in cots c++ binaries. In *Proceedings of the Network and Distributed System Security Symposium*, 2015.

[42] Majid Salehi, Danny Hughes, and Bruno Crispo. μSBS: Static binary sanitization of bare-metal embedded devices for fault observability. In *Proceedings of the International Conference on Research in Attacks, Intrusions, and Defenses*, pages 381–395, 2020.

[43] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the USENIX Annual Technical Conference*, pages 309–318, 2012.

[44] Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. Security analysis of processor instruction set architecture for enforcing control-flow integrity. In *Proceedings of the International Workshop on Hardware and Architectural Support for Security and Privacy*, 2019.

[45] Ubuntu wiki. Compilerflags. https://wiki.ubuntu.com/ToolChain/CompilerFlags.

[46] Freek Verbeek, Nico Naus, and Binoy Ravindran. Verifiably correct lifting of position-independent x86-64 binaries to symbolized assembly. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2024.

[47] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. Ramblr: Making reassembly great again. In *Proceedings of the Network and Distributed System Security Symposium*, 2017.

[48] Shuai Wang, Pei Wang, and Dinghao Wu. Reassembleable disassembling. In *Proceedings of the USENIX Security Symposium*, pages 627–642, 2015.

[49] Xinran Wang, Yoon-Chan Jhi, Sencun Zhu, and Peng Liu. Behavior based software theft detection. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 280–290, 2009.

[50] Richard Wartell, Vishwath Mohan, Kevin W Hamlen, and Zhiqiang Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 157–168, 2012.

[51] Richard Wartell, Vishwath Mohan, Kevin W Hamlen, and Zhiqiang Lin. Securing untrusted code via compiler-agnostic binary rewriting. In *Proceedings of the Annual Computer Security Applications Conference*, pages 299–308, 2012.

[52] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P. Kemerlis. Egalito: Layout-agnostic binary recompilation. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 133–147, 2020.

[53] Yuan Xiao, Mengyuan Li, Sanchuan Chen, and Yinqian Zhang. STACCO: Differentially analyzing side-channel traces for detecting SSL/TLS vulnerabilities in secure enclaves. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 859–874, 2017.

[54] Junyuan Zeng, Yangchun Fu, Kenneth A Miller, Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Obfuscation resilient binary code reuse through trace-oriented programming. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 487–498, 2013.

[55] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, László Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 559–573, 2013.

[56] Mingwei Zhang, Rui Qiao, Niranjan Hasabnis, and R. Sekar. A platform for secure static binary instrumentation. In *Proceedings of the International Conference on Virtual Execution Environments*, pages 129–140, 2014.

[57] Mingwei Zhang and R. Sekar. Control flow integrity for COTS binaries. In *Proceedings of the USENIX Security Symposium*, pages 337–352, 2013.

A Artifact Appendix

A.1 Abstract

The artifact contains the code and datasets used in our experiments, along with scripts to reproduce the results presented in our evaluation. Specifically, it includes: (a) Coreutils-9.1 and Binutils-2.40 binaries used in our tests; (b) scripts for generating SPEC CPU 2006 and SPEC CPU 2017 binaries; (c) scripts for generating rewritten binaries with SURI, Ddisasm, and Egalito; (d) scripts for running the test suite to evaluate reliability; (e) scripts for measuring SURI's overhead; and (f) scripts for demonstrating realworld applications; and (e) detailed instruction documentation for using SURI. Everything is packaged and pre-built as a Docker image. We expect a standard x86-64 Linux machine with Docker and Python 3 installed to run this artifact.

Our artifact does *not* include SPEC CPU 2006 nor SPEC CPU 2017 binaries due to license restrictions. However, we provide scripts to generate SPEC binaries to help fully reproduce our results in case the reader has a valid SPEC license and their source code.

A.2 Artifact check-list (meta-information)

- **Program:** .NET 7.0, Python 3 with pip installed, gcc-11 is required to run SURI locally. For this artifact evaluation, Python 3 and Docker are necessary.
- **Data set:** Coreutils (v9.1), Binutils (v2.40), SPEC CPU 2006 (v1.2), SPEC CPU 2017 (v1.1.5). For SPEC CPU benchmarks, please see A.3.4.
- **Run-time environment:** Linux
- **Hardware:** x86-64 Machine
- **Output:** Evaluation results of reassembly on our dataset, reproducing Table 2 to Table 5 in our paper.
- **How much disk space required (approximately)?:** Running experiments only on the Coreutils and Binutils provided in the artifact dataset requires about 500GB. If including SPEC benchmark programs by building them, approximately 3TB is needed.
- **How much time is needed to prepare workflow (approximately)?:** Using the provided dataset, the setup with Docker takes around 2–3 hours. If including SPEC benchmarks, additional time for building and generating metadata is required, bringing the total preparation time to approximately 5–6 days.
- **How much time is needed to complete experiments (approximately)?:** Running experiments with the basic benchmark binaries takes about 7 days. If including the SPEC benchmarks, the total runtime increases to approximately 30 days.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** MIT License
- **Archived (provide DOI)?:** On Zenodo (https://doi.org/10.5281/zenodo.14788616)

A.3 Description

A.3.1 How to access. Download the artifact from the Zenodo (https://doi.org/10.5281/zenodo.14788616), or GitHub (https://github.com/SoftSec-KAIST/SURI).

A.3.2 Hardware dependencies. A standard x86-64 machine with more than 64GB of RAM is required. The Docker instance used in the artifact is allocated 64GB of memory. For our experiments, we used a machine equipped with an Intel Core i9-11900K processor and 128GB of RAM.

A.3.3 Software dependencies. Our artifact is based on Linux system. If you want to run SURI locally, .NET 7.0, Python 3, and gcc-11 should be installed on your system. For the artifact evaluation, the minimal software requirement is Docker and Python 3.

A.3.4 Data sets. In our experiments, total five different kinds of benchmark binaries are used: Coreutils-9.1, Binutils-2.40, SPEC CPU 2006 and 2017, 10 real-world programs, and Juliet test suite.

Our artifact includes prebuilt Coreutils and Binutils binaries as well as metadata containing information about their assembly code. Moreover, our dataset also contains prebuilt real-world programs and Juliet test suite binaries. We did not include SPEC CPU 2006 and SPEC CPU 2017 binaries in the artifact because of the license restrictions, but readers can generate them using the provided scripts if they have a valid SPEC license.

For more details about our dataset, please refer to our README.md for the artifact evaulation.

A.4 Installation

To install SURI on your local environment, execute the following commands:

```
$ python3 setup.py install -user
$ cd superCFGBuilder
$ dotnet build
```

If you want to use SURI with Docker environment, execute the following commands:

```
$ python3 setup.py install -user
$ docker build -t suri:v1.0 .
```

A.5 Experiment workflow

A.5.1 Preparation. To reproduce our evaluation results, you need to prepare the experimental environments first. Our PREPARATION.md explains how you can download our artifact, how to build Docker images for the evaluation workflows, and how to build SPEC benchmark binaries in case you have a valid license of SPEC CPU benchmark.

A.5.2 Experiment. We provide five experiments to reproduce our evaluation results.

1. Reassembly completion comparison (RQ1).
2. Test suite pass rate comparison (RQ1).
3. Reliability test on real-world programs (RQ1).
4. Overhead of rewritten binaries (RQ2).
5. Application of SURI (RQ3).

For more detailed information, please refer to EXPERIMENT.md.

A.6 Evaluation and expected results

The first two experiments reproduce Table 2 and 3, as well as results shown on Section 4.3.3 in our paper. The third experiment reproduces results explained on Section 4.2.3 in our paper. The fourth experiment reproduces Section 4.3 including Table 4, and the last experiment reproduces Table 5.

A.7 Methodology

Submission, reviewing, and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-and-badging-current
- https://cTuning.org/ae