Reassembly is Hard: A Reflection on Challenges and Strategies

Hyungseok Kim[1,2], Soomin Kim[1], Junoh Lee[1], Kangkook Jee[3], and Sang Kil Cha[1]
[1]*KAIST* [2]*The Affiliated Institute of ETRI* [3]*University of Texas at Dallas*
{witbring,soomink,junoh,sangkilc}@kaist.ac.kr kangkook.jee@utdallas.edu

Abstract

Reassembly, a branch of static binary rewriting, has become a focus of research today. However, despite its widespread use and research interest, there have been no systematic investigations on the techniques and challenges of reassemblers. In this paper, we formally define different types of errors that occur in current existing reassemblers, and present an automated tool named REASSESSOR to find such errors. We attempt to show through our tool and the large-scale benchmark we created the current challenges in the field and how they can be approached.

1 Introduction

Static binary rewriting is imperative to software security in its ability to inline security monitors to binaries without access to the source code. This technique is often used to harden legacy binaries by ensuring Control Flow Integrity (CFI) [29,98,102] or by randomizing code layouts [20, 41, 46, 61, 87, 92, 98]. It has also been well developed in other domains such as malware analysis [12, 24, 45, 97], software debloating [65], bug finding [26, 57], and automated code repair [72].

Despite the surging research interests, however, current state-of-the-art techniques still suffer from either applicability or performance overhead. Patch-based approaches, such as Detour [35], Bistro [24], and E9Patch [27], incur little overhead but are limited in the scope of instrumentation points. Table-based approaches such as PSI [100], Multiverse [5], and μSBS [71], have no such limit but impose both a time and space overhead.

In contrast, reassembly [26,32,88,89,95] is a recent attempt in static binary rewriting to remedy both of these problems. It allows an analyst to add instrumentation to any point in the target binary while keeping both the time and space overhead to a minimum. The key insight here is to first translate a binary into a *relocatable* Intermediate Representation (IR), where instructions can be re-positioned without having to modify their syntax [63]. For example, disassembly instructions involve hard-coded addresses or offsets, whereas IRs would contain symbolic labels to refer to such addresses. Therefore, such IRs can be easily instrumented with an inline monitor and compiled back to produce a rewritten binary.

To build a relocatable IR from a binary, however, one needs to be able to recover the cross-references in the binary. This is often referred to as a *symbolization* challenge. At a high level, symbolization is the process of restoring symbolic labels, used to make a cross-reference in the IR, from the numeric values in the target binary. Symbolization is challenging because (1) one needs to first identify which numbers from the binary to symbolize, and (2) the numbers in the binary are often formed by a compound symbolic expression.

For instance, consider the following instruction "`push 0x42424242`", where `0x42424242` is the address of a global variable `foo`. When the instruction is given without any further information, we cannot simply determine that the number refers to the address of `foo`: It can be merely a constant literal used in the program. The problem only exacerbates when the binary dynamically computes such addresses at runtime.

For these reasons, reassembly has been limited to small size binaries with predictable control references. Although several heuristics-based solutions have been proposed [32, 88, 89], they all suffer from the imprecision of the underlying symbolization technique: They often mistakenly identify a literal as a pointer or vice versa.

Nonetheless, reassembly is gaining substantial attention especially with increasing use of Position Independent Executable (PIE) binaries. PIEs use relative addressing modes, such as Program Counter (PC)-relative or Global Offset Table (GOT)-relative addressing, and make a relocation table entry in the binary for handling absolute addresses. Therefore, reassemblers do not need to distinguish absolute addresses from constant literals for PIEs, making it seemingly easier than non-PIEs. Indeed, the authors of RetroWrite even claim that their tool can *soundly* rewrite PIEs without the precise recovery of the Control-Flow Graph (CFG) [26].

However, such emerging research trends in reassembly could possibly give a false impression of the field because position-independence itself cannot be a solution to the symbolization challenge as other researchers have also noted [32]. Notably, compiler-generated values, such as jump table entries, do not always have relocation information, making it difficult to recover the original symbolic labels. Furthermore, imprecise disassembly can cause various reassembly failures as well as symbolization errors.

In this paper, we systematically analyze such problems with our tool, named REASSESSOR. We first formally define several different errors that occur in each reassembler. We then design and implement REASSESSOR to identify them. At a high level, REASSESSOR finds reassembly errors by diffing compiler-generated assembly code and reassembler-generated assembly code. Note that reassemblers are widely known to have symbolization errors [32,88], but there have been limited attempts at systematically finding them.

We ran REASSESSOR on the benchmark consisting of 14,688 binaries compiled with various compilers and compiler options. With our tool and benchmark, we found that none of the existing reassemblers is free from symbolization errors, and we were able to create a meaningful patch to one of those tools, too. These results show the current challenges in reassembly and provide guidance for future research. In summary, we make the following contributions:

- We propose a formal framework to classify reassembler errors into eight categories.
- We demonstrate REASSESSOR, an automated tool for finding the defined errors from reassemblers.
- We present a thorough benchmark for evaluating reassemblers.
- We identify various real-world reassembly errors from state-of-the-art tools and summarize lessons learned.
- We publicize our tool as well as our benchmark to foster future research: `https://github.com/SoftSec-KAIST/Reassessor`

2 Reassembly

In this section, we first clarify several terms including reassembly and symbolization. We then formally define symbolization errors and categorize different error types.

2.1 Reassembly and Symbolization

The term "reassembly" was first introduced in 2015 by Uroboros [89]. At a high level, reassembly is a static binary rewriting process that works by transforming a binary into a *relocatable representation* such as an Intermediate Representation (IR) or an assembly. The relocatable form can then be trivially instrumented and compiled (or assembled) back to a rewritten binary.

To create a relocatable representation, reassemblers need to first analyze which parts in the binary code denote a reference and turn these references into a symbol. We call such a step the symbolization process. Note that reassembly is different from binary lifting because binary lifting does not involve the symbolization process [39,44].

The idea of translating a binary into an intermediate form and then recompiling it back to a binary dates back to the 1980s [52]. Traditionally, we call such a technique as binary translation [78], which mainly focused on the *cross-architecture* retargetability, i.e., ISA-to-ISA translation [21,76,91,103]. Previous static binary translators relied on a specific run-time environment, often referred to as a fallback mechanism, to handle difficult-to-analyze cases such as indirect jumps [22,23,83].

One might view reassembly then as a way to achieve *fully static* binary translation that does not rely on any runtime support. Although there has been a substantial body of work on static binary translation, such as SecondWrite [58,79], LLBT [77], McSema [25], and Zipr [33], they do not fully leverage symbolization, by either limiting their instrumentation capabilities or relying on runtime support. In this paper, we use the term *reassembly* to exclusively mean a fully static binary translation technique that satisfies the followings:

1. The technique should not rely on runtime support. For example, we do not regard BinRec [1] as a reassembler because it operates on execution traces.
2. The technique should use a symbolization approach when generating a relocatable representation.

2.2 Symbolization Error

During a symbolization process, reassemblers may miss some labels to symbolize, turn some immediate values into wrong labels, or even falsely symbolize some constant literals although they should never be symbolized. We call such an error "*symbolization error*", and formally define it after introducing several terms and assumptions.

Assembly File (α). For brevity, we assume that both compilers and reassemblers produce only a single assembly file α per program. Even if a tool produces multiple assembly files in practice, we can simply combine them to form a single file. We further assume that assembly files are in the Intel syntax.

Assembly and Reassembly Processes. Let α_c be an assembly file obtained from a compiler, and let β be the binary obtained by assembling α_c. We denote the assembly process by $\texttt{Asm}$. That is, $\texttt{Asm}(\alpha_c) = \beta$. We then let α_r be the assembly file obtained by reassembling β without adding any instrumentation. We use $\texttt{Reasm}$ to denote the reassembly process: $\texttt{Reasm}(\beta) = \alpha_r$. Figure 1 illustrates the relationships between α_c, α_r, and β. To detect symbolization errors, we analyze the difference of the labels in α_c and α_r.

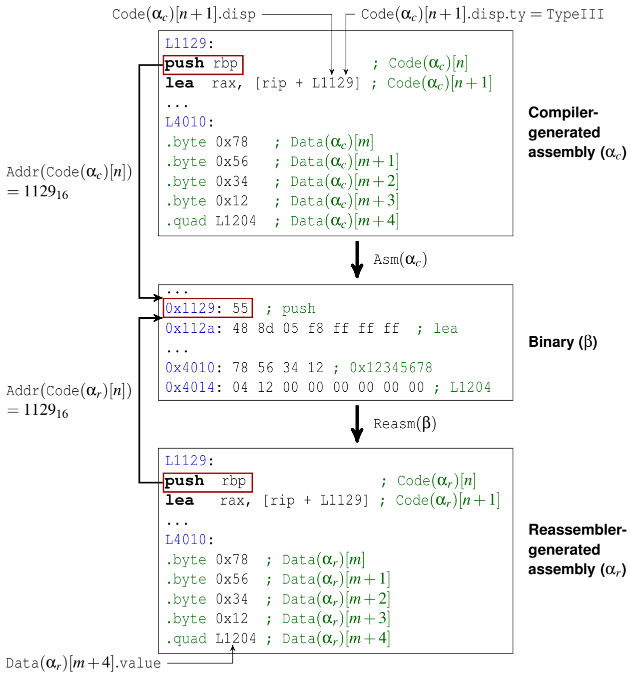


Figure 1: Visual description of symbols used.

Normalization. To ease the comparison between α_c and α_r, we assume that both α_c and α_r are normalized to satisfy the following criteria. First, every assembly label should start with the prefix 'L' followed by its address in β. For example, in Figure 1, the label in Line n of α_r is normalized to `L1129` as its corresponding address in β is `0x1129`. For those labels with a special suffix, such as `@GOTOFF`, we preserve the suffix while normalizing the main part. Second, any numbers in an assembly file, whether they are from code or data, should be represented in hexadecimal notation. Finally, every concrete value declared in a data section should be one-byte long. While a `long` integer `0x12345678` can be defined as "`.long 0x12345678`", our normalization process will break it into

```
type relocexpr_type = TypeI | TypeII | ... | TypeVII

type relocexpr = { // Relocatable expression.
  str: string, // String representation of the expr.
  ty: relocexpr_type  // Relocatable expression type.
}

type instruction = {
  str: string,       // Assembly instruction string.
  displ: relocexpr, // displacement or null.
  imm: relocexpr    // immediate or null.
}

type dataline = {
  value: relocexpr  // data value or null.
}
```

Figure 2: ML-style types used in our formal framework.

Table 1: Categorization of relocatable expressions.

		Syntax	
		Atomic	**Composite**
Semantics	**Absolute address**	Type I	Type II
	PC-relative address	Type III	Type IV
	GOT-relative address	Type V	Type VI
	Label-relative address	-	Type VII

four consecutive one-byte values as shown in our example (see the label `L4010`). Note, however, data declarations with a symbolic expression (e.g., the lines that start with `.quad` in our example) will not be partitioned.

Relocatable Expression. Assembly code is relocatable as any addresses or relative offsets are denoted by a symbolic expression, which will be called a relocatable expression. For example, the PC-relative offset of the `lea` instruction in Figure 1 is shown as a relocatable expression `L1129`. More formally, a relocatable expression in an assembly file is a symbolic expression with one or more labels, which will be eventually translated into a number, e.g., an immediate or a displacement, in the corresponding binary. In this paper, we represent a relocatable expression as a record (`relocexpr`) as defined in Figure 2. The `ty` field of `relocexpr` returns a relocatable expression type `relocexpr_type`, which is used to distinguish relocatable expressions based on their syntactic and semantic properties as shown in Table 1.

1. **Syntax-based Classification**. We say a relocatable expression is *atomic* if it solely consists of a single label, and *composite* if it is represented with a compound expression. For example, in Figure 3, `.LBB0_1` is an atomic expression, whereas `msg+16` is a composite, which is translated into the displacement `0x200a06` in the binary shown in Figure 3c. However, it is difficult to recover the original relocatable expression by merely looking at the displacement. Moreover, composite relocatable expressions are present in most binaries: 97.4% of the binaries in our benchmark (§5.2.3).

2. **Semantics-based Classification**. We also distinguish relocatable expressions based on their semantics about how they are used to compute an Effective Address (EA). In Intel, there are four different ways to compute an EA. First, one may use an absolute address to directly refer to an EA. One can also obtain an EA in relation to a base point, where the base point is (1) the current Program Counter (PC), (2) the Global Offset Table (GOT), or (3) an arbitrary label other than GOT. Among the three cases, we found that label-relative offsets always have the form of "$\texttt{label}_1$ (op) $\texttt{label}_2$", where (op) is a binary operator. Thus, they can only be a composite.

Accessing Code. Let `Code` be a function that takes in an assembly file f as input and returns an array of instructions in f as output. Each instruction is a record (`instruction`) defined in Figure 2. In Intel assembly instructions, relocatable expressions can only appear as a displacement (`disp`) or as an immediate (`imm`).[1] Thus, the `instruction` record has two dedicated fields to help access relocatable expressions. Note we do not need to distinguish between operands here as there can be at most one displacement and one immediate per instruction in Intel [36]. Both the fields are *nullable*, meaning that they can return a `null` when there is no displacement/immediate in the instruction or the instruction has a constant displacement/immediate, i.e., no symbolic expression. In Figure 1, for example, we can access the displacement of the mth instruction of α_c with $\texttt{Code}(\alpha_c)[m].\texttt{disp}$.

Accessing Data. Similarly, let `Data` be a function that takes in an assembly file and outputs an array of assembly lines that are associated with a data value. We call such assembly lines a data line (`dataline` type in Figure 2). We access the value of a data line with the `value` field, which returns a relocatable expression (`relocexpr`) if it exists. It will return `null` when the data line has a constant value. In Figure 1, for instance, $\texttt{Data}(\alpha_r)[m].\mathit{value} = \texttt{null}$ and $\texttt{Data}(\alpha_r)[m+4].\mathit{value} = \texttt{L1204}$.

Accessing Addresses. We let `Addr` be a function that takes in either an `instruction` or a `dataline` as input, and returns the corresponding address in β. This function makes explicit the relationship between two assembly lines respectively in α_c and α_r by referring to the address in the binary β. The red boxes in Figure 1 shows that `Addr` returns the address `0x1129` for both $\texttt{Code}(\alpha_c)[n]$ and $\texttt{Code}(\alpha_r)[n]$.

Symbolization Error. A symbolization error occurs when two assembly lines respectively in α_c and α_r have a difference in their labels while representing the same instruction or data value in β. We now define it formally as follows.

Definition 1 (Symbolization Error)**.** Given α_c and $\alpha_r = \texttt{Reasm}(\texttt{Asm}(\alpha_c))$, `Reasm` has a symbolization error if and only if there exist m and n such that

$$
\begin{array}{l}
\left(\begin{array}{ll}
 & \texttt{Addr}(\texttt{Code}(\alpha_c)[m]) = \texttt{Addr}(\texttt{Code}(\alpha_r)[n]) \\
\wedge & \texttt{Code}(\alpha_c)[m] \neq \texttt{Code}(\alpha_r)[n]
\end{array}\right) \\
\vee \left(\begin{array}{ll}
 & \texttt{Addr}(\texttt{Data}(\alpha_c)[m]) = \texttt{Addr}(\texttt{Data}(\alpha_r)[n]) \\
\wedge & \texttt{Data}(\alpha_c)[m] \neq \texttt{Data}(\alpha_r)[n]
\end{array}\right).
\end{array}
$$

Symbolization errors can be divided into two cases: false positives and false negatives. We say there is a False-Negative (FN) error when the reassembler fails to recover a relocatable expression from a number in $\beta = \texttt{Asm}(\alpha_c)$, while the corresponding assembly line in α_c has a relocatable expression.

[1] A displacement is a number in a memory operand, e.g., 42 in `mov rax, [rdx + 42]`. An immediate is a number-only operand, e.g., 42 in `push 42`.

```c
char msg[] = "Hi Reassembler\n";
void foo()
{
  for(char *p = msg; p < msg+sizeof(msg); ++p)
    putchar(*p);
}
```

(a) Source code in C.

```
.section   .text
foo:
  push    r14
  push    rbx
  push    rax
  lea     rbx, [rip+msg]
  lea     r14, [rip+msg+16]
.LBB0_1:
  movsx   edi, byte ptr [rbx]
  xor     eax, eax
  call    putchar@PLT
  inc     rbx
  cmp     rbx, r14
  jb      .LBB0_1
  add     rsp, 8
  pop     rbx
  pop     r14
  ret

.section   .data
msg:
.asciz "Hi Reassembler\n"
```

(b) x86-64 assembly code produced by Clang.

```
Disassembly of section .text:
0x628:  push   r14
0x62a:  push   rbx
0x62b:  push   rax
0x62c:  lea    rbx, [rip+0x2009fd]
0x633:  lea    r14, [rip+0x200a06]
0x63a:  movsx  edi, BYTE PTR [rbx]
0x63d:  xor    eax, eax
0x63f:  call   520
0x644:  inc    rbx
0x647:  cmp    rbx, r14
0x64a:  jb     63a
0x64c:  add    rsp, 0x8
0x650:  pop    rbx
0x651:  pop    r14
0x653:  ret
  ; ...
Contents of section .data
0x201030: 48 69 20 ... ; "Hi Reassemblr"
  ; ...
Contents of section .bss
0x201040: 00 00 00 00 ...
```

(c) Disassembled PIE binary code.

Figure 3: Example describing a symbolization challenge.

Definition 2 (False Negatives)**.** Given α_c and $\alpha_r = \texttt{Reasm}(\texttt{Asm}(\alpha_c))$, `Reasm` has a false-negative error if and only if there exist m and n such that

$$
\begin{array}{l}
\left(\begin{array}{ll}
 & \texttt{Addr}(\texttt{Code}(\alpha_c)[m]) = \texttt{Addr}(\texttt{Code}(\alpha_r)[n]) \\
\wedge & \texttt{Code}(\alpha_c)[m].\texttt{disp} \neq \texttt{null} \\
\wedge & \texttt{Code}(\alpha_r)[n].\texttt{disp} = \texttt{null}
\end{array}\right) \\
\vee \left(\begin{array}{ll}
 & \texttt{Addr}(\texttt{Code}(\alpha_c)[m]) = \texttt{Addr}(\texttt{Code}(\alpha_r)[n]) \\
\wedge & \texttt{Code}(\alpha_c)[m].\texttt{imm} \neq \texttt{null} \\
\wedge & \texttt{Code}(\alpha_r)[n].\texttt{imm} = \texttt{null}
\end{array}\right) \\
\vee \left(\begin{array}{ll}
 & \texttt{Addr}(\texttt{Data}(\alpha_c)[m]) = \texttt{Addr}(\texttt{Data}(\alpha_r)[n]) \\
\wedge & \texttt{Data}(\alpha_c)[m].\texttt{value} \neq \texttt{null} \\
\wedge & \texttt{Data}(\alpha_r)[n].\texttt{value} = \texttt{null}
\end{array}\right).
\end{array}
$$

Similarly, we say there is a False-Positive (FP) error when the reassembler recovered a wrong relocatable expression from the given binary $\beta = \texttt{Asm}(\alpha_c)$.

Definition 3 (False Positives)**.** Given α_c and $\alpha_r = \texttt{Reasm}(\texttt{Asm}(\alpha_c))$, `Reasm` has a false-positive error if and only if there exist m and n such that

$$
\begin{array}{l}
\left(\begin{array}{ll}
 & \texttt{Addr}(\texttt{Code}(\alpha_c)[m]) = \texttt{Addr}(\texttt{Code}(\alpha_r)[n]) \\
\wedge & \texttt{Code}(\alpha_c)[m].\texttt{disp} \neq \texttt{Code}(\alpha_r)[n].\texttt{disp} \\
\wedge & \texttt{Code}(\alpha_r)[n].\texttt{disp} \neq \texttt{null}
\end{array}\right) \\
\vee \left(\begin{array}{ll}
 & \texttt{Addr}(\texttt{Code}(\alpha_c)[m]) = \texttt{Addr}(\texttt{Code}(\alpha_r)[n]) \\
\wedge & \texttt{Code}(\alpha_c)[m].\texttt{imm} \neq \texttt{Code}(\alpha_r)[n].\texttt{imm} \\
\wedge & \texttt{Code}(\alpha_r)[n].\texttt{imm} \neq \texttt{null}
\end{array}\right) \\
\vee \left(\begin{array}{ll}
 & \texttt{Addr}(\texttt{Data}(\alpha_c)[m]) = \texttt{Addr}(\texttt{Data}(\alpha_r)[n]) \\
\wedge & \texttt{Data}(\alpha_c)[m].\texttt{value} \neq \texttt{Data}(\alpha_r)[n].\texttt{value} \\
\wedge & \texttt{Code}(\alpha_r)[n].\texttt{value} \neq \texttt{null}
\end{array}\right).
\end{array}
$$

Table 2: Categorization of symbolization errors.

ID	Relocatable Expression in α_c			Observable				FP/FN	Ex.
	Syntax	Semantics	Type	32	64	PIE	noPIE		
E1	Atomic	Absolute	I	✓	✓	✓	✓	FP FN	§A.1 §A.2
E2	Composite	Absolute	II	✓	✓	✓	✓	FP FN	§A.3 §A.4
E3	Atomic	PC-rel	III	✓	✓	✓	✓	FP FN	§A.5 §A.6
E4	Composite	PC-rel	IV	✗	✓	✓	✓	FP FN	§A.7 §A.8
E5	Atomic	GOT-rel	V	✓	✗	✓	✗	FP FN	§A.9 §A.10
E6	Composite	GOT-rel	VI	✓	✗	✓	✗	FP FN	§A.11 §A.12
E7	Composite	Lab-rel	VII	✓	✓	✓	✗	FP FN	§A.13 §A.14
E8	Constant	-	-	✓	✓	✓	✓	FP	§A.15

2.3 Categorization of Symbolization Errors

Recall from §2.2, a symbolization error occurs when there is a mismatch between two corresponding relocatable expressions (`relocexpr`) respectively in α_c and α_r. We can further categorize symbolization errors based on the properties of the mismatched relocatable expressions.

Suppose there is a mismatch between two relocatable expressions $e_c \in \alpha_c$ and $e_r \in \alpha_r$. We can then classify symbolization errors into the eight categories based on the type of e_c, as shown in Table 2. In case e_c is `null`, the error is always due to the false symbolization of a non-relocatable expression. Thus, we separately consider this case as **E8**. We further subdivide each error category based on whether they are a False Positive (FP) or a False Negative (FN). This gives us a total of fifteen different error cases, because **E8** can only have false positives by definition. For each of the error categories, we present in the Appendix an example error case that REASSESSOR found as indicated by the last column of Table 2. The Observable column in the table summarizes whether each of the error types is observed in our benchmark.

3 REASSESSOR Design

This section describes the design and implementation of REASSESSOR, an automated tool for detecting symbolization errors defined in §2.3. We start by introducing the overall architecture of REASSESSOR and describe the design challenge of REASSESSOR. We then present the details of each module and show how we address the challenges. Finally, we discuss the soundness of our system as well as the implementation details of REASSESSOR.

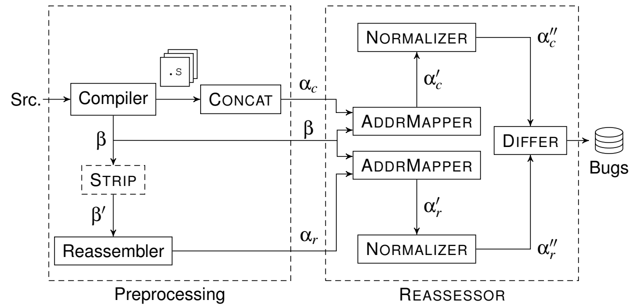


Figure 4: REASSESSOR architecture.

3.1 Overview

At a high level, REASSESSOR takes in as input a compiler-generated assembly file (α_c), a binary file (β), and a reassembler-generated assembly file (α_r). It then outputs a list of symbolization errors found. Figure 4 depicts the overall architecture of REASSESSOR.

First, there is a preprocessing step that needs to be performed before operating REASSESSOR, which is to run both a compiler and a reassembler under test to produce a triple (α_c, β, α_r). The CONCAT module merges all the assembly files generated by the compiler into one. The STRIP module strips off debug symbols from the binary β to get a stripped binary β'. The stripping process is omitted for some reassemblers if they require debugging information to operate, e.g., RetroWrite. We further detail the preprocessing step in §3.2.

Next, the ADDRMAPPER module takes in an assembly file and a non-stripped binary β as input, and returns an annotated assembly file that provides means to identify the corresponding addresses of the assembly lines. That is, it parses the given assembly file and maps each line in the assembly file with a concrete address appeared in the given binary. Given the triple (α_c, β, α_r), we run ADDRMAPPER twice with two different inputs: (α_c, β) and (α_r, β). This way we can obtain two annotated assembly files: α_c' and α_r'. §3.3 details the design of ADDRMAPPER. Each of the annotated assembly files then goes through the NORMALIZER module, which transforms assembly expressions into a canonical form to ease the comparison. In return, we obtain normalized (and annotated) assembly files: α_c'' and α_r''. We describe the detailed implementation in §3.4.

Finally, the DIFFER module takes in the two normalized assembly files (α_c'' and α_r'') as input, and returns a list of symbolization errors found. In our implementation, DIFFER also reports reassembly bugs that are not a symbolization error. For example, it can also detect reassembly bugs that are due to erroneous disassembly. §3.5 details its design.

Challenges. There are several technical challenges in designing REASSESSOR. First, obtaining assembly files during compilation is not always straightforward due to complex source file structures (§3.2). Second, reassemblers can produce grammatically wrong assembly files as output due to implementation errors (§3.3.1). Third, there can be multiple matching assembly lines for a single disassembled instruction (§3.3.2). Finally, not every assembly line has an associated debugging symbol (§3.3.3).

3.2 Generating Assembly Files

Most modern compilers provide a command line switch (such as `-save-temps`) that forces the compilers to preserve all the intermediate files including assembly files generated during a compilation process. Although it seems trivial, obtaining assembly files from a compiler is challenging due to potentially complex source structures.

Suppose there are two programs that share a source file f, which contains `#if` directives to provide two or more different implementations of the same function in f. When the two programs define different macros, we will obtain two different versions of assembly files from f for each program. Unfortunately, those two assembly files share the same path because they are from the same source file. Thus, if we compile the package with the `make` command, one assembly file will overwrite the other, leaving only one assembly file. We observed this problem in the GNU coreutils package, and Clang was not able to separate assembly files in this case.

To handle the aforementioned challenge, we leverage `loggedfs` [31] while building a project. It allows us to check if any assembly file has been overwritten by the compiler. When we identify such cases during compilation, we manually fixed the corresponding `Makefile`(s) to retrieve assembly file(s).

3.3 Address Mapping

Recall that ADDRMAPPER associates concrete addresses in β with assembly lines in α to produce α', which is an annotated assembly file that has a mapping from each assembly line to its address. This is to implement the `Addr` function defined in §2.2. There are two design requirements that need to be satisfied for ADDRMAPPER. First, our tool should be resilient to parsing errors because reassemblers often produce grammatically incorrect assembly files (§3.3.1). Second, our tool should be able to identify concrete addresses for assembly lines located in both code (§3.3.2) and data sections (§3.3.3).

3.3.1 Error-Resilient Parsing

Reassemblers sometimes produce grammatically wrong assembly files due to implementation errors. If we simply regard such cases as a bug, we will not be able to figure out the actual symbolization problems thwarting the *reassembly* process.

During the course of our study, we found that Ramblr, RetroWrite, and Ddisasm can generate invalid assembly files including ones with (1) duplicate label definitions, and (2) references to undefined labels. Therefore, we implemented our own assembly parser, which can disregard such parsing errors and keep consuming the next assembly lines.

3.3.2 Calculating Code Addresses

Compilers often produce duplicate function bodies in different assembly files, but only one of them will be selected when emitting a binary. Furthermore, each duplicate copy may have slightly different instructions due to the use of C macros. Therefore, ADDRMAPPER should be able to identify the right function in α for a function in β. We handle this challenge by comparing instruction sequences.

Specifically, we associate the address in β with every assembly instruction in α in the following three steps. First, we enumerate every function in β with the help of the debugging information. Second, for each function, we find all possible functions in α. Third, for each function found in the previous step, we identify a matching function in β by comparing their instructions. We then assign concrete addresses to the function and its instructions only when there is a match.

While matching functions, we carefully consider compiler-generated no-op instructions, which exist only in β, but not in α. Such no-op instructions have many different forms, e.g., "`nop`", "`nop DWORD ptr [eax+eax*1+0x0]`", "`lea esi, [esi]`", and so forth. REASSESSOR regards every instruction that does not change the CPU state other than the PC register as a "semantic no-op instruction", and ignores them to correctly match every function.

3.3.3 Calculating Data Addresses

Unlike instruction addresses, not every data value in a binary has a debug symbol attached to it. For example, compiler-generated data values, such as jump table entries, have no debug symbol. Therefore, one cannot simply adopt the same method we used for obtaining code addresses.

At a high level, ADDRMAPPER uses two different methods to compute data addresses: (1) for compiler-generated assembly files, it examines local symbols generated by the compiler; and (2) for reassembly-generated assembly files, it leverages tool-specific metadata generated by each reassembler.

Data Addresses for α_c. Compilers assign a *local symbol* to compiler-generated data values, which is easily identifiable as they are always prefixed by a dot (`.`) symbol. Furthermore, we can infer data addresses by examining how local symbols are referenced in the assembly file (α_c) as illustrated in Figure 5. First, it enumerates all possible local symbols (including the symbol `.Lswitch.table.convert_move`). Next, for each local symbol, it searches for an instruction that references the symbol. Finally, ADDRMAPPER locates the corresponding instruction in β with the debugging informa-

```
; code section
.Ltmp516:
mov  eax, [eax * 4 + .Lswitch.table.convert_move]
jmp  .LBB8_169
...
; data section
.Lswitch.table.convert_move: 1
4 .long libfunc_table
  .long libfunc_table+4
  .long libfunc_table+8
...
```

Compiler-generated assembly α_c

```
...
0x80d226e: mov eax, [eax * 4 + 0x823800c]
...
```

Corresponding binary β

Figure 5: Calculating data addresses from local symbols.

tion. We note that the corresponding displacement value of `.Lswitch.table.convert_move` is `0x823800c`. Hence, we realize that the data line 4 has the value `0x823800c`.

Data Addresses for α_r. Any data values that are examined by the reassembler are explicitly assigned with a label. Existing reassemblers that we studied always produce an assembly label with enough metadata attached to it for debugging purposes. For example, every data value in α_r generated by Ramblr has an explicit annotation showing at which address the data value is located. Thus, ADDRMAPPER parses such meta information to construct a mapping from a data line to the binary address.

3.4 Assembly Normalization

Recall from §2.2, our definition of symbolization error is based on the assumption that assembly files are syntactically normalized. In our implementation, NORMALIZER converts an annotated assembly file α' into another annotated assembly file α'', which contains only canonical assembly expressions making a comparison between assembly files straightforward.

Specifically, NORMALIZER first parses an assembly file written in either the AT&T syntax or the Intel syntax into a data structure representing the Abstract Syntax Tree (AST) of the assembly file. It then converts labels in the AST to have a normalized name with the corresponding address (as described in §2.2). Next, NORMALIZER breaks constant data values into a sequence of byte values. The modified ASTs will then be used as input to the DIFFER module.

3.5 Assembly Diffing

The last step of REASSESSOR is DIFFER, which compares two annotated assembly files α''_c and α''_r to find potential errors in the reassembler under test, i.e., `Reasm`. Specifically, DIFFER compares the ASTs of the assembly files, and sees if there is any mismatch. Note DIFFER ignores compiler-generated functions and sections for diffing. For every mismatch found, it examines the mismatched expression in both α''_c and α''_r to decide the error type, and reports the error. As an example, consider the error case in §A.2 where there is a mismatch in the second operand of the `cmp` instructions. In this case, REASSESSOR will realize that the atomic relocatable expression `L759ab0` is not symbolized by the reassembler under test. Since the expression represents an absolute address, it is a Type I relocatable expression, and this is a false-negative error. Therefore, REASSESSOR will report this error as an **E1** false-negative error according to Table 2.

In our current implementation, REASSESSOR detects not only symbolization errors, but also disassembly errors. It is indeed straightforward to identify disassembly errors by comparing two AST expressions. Our study confirms that current reassemblers suffer from disassembly errors, too (§5.3.2).

3.6 Soundness of REASSESSOR

Any symbolization errors found by REASSESSOR can potentially break the program semantics as long as the erroneous program point is reachable. For instance, if there is a symbolization error in an unreachable instruction, then the error will give no harm to the program behavior. However, we believe such unsound cases are rare in practice due to various compiler optimization techniques, such as dead code elimination. It is beyond the scope of this paper to verify whether a program point is reachable or not.

3.7 Implementation

We have implemented REASSESSOR in approximately 3.1K SLoC of Python: 0.3K SLoC for the preprocessing module, 2.8K SLoC for the main modules (ADDRMAPPER, NORMALIZER, and DIFFER) of REASSESSOR. We leveraged Capstone [67] for disassembling binaries, and pyelftools [7] for parsing ELF headers and DWARF debugging information.

4 Building Benchmark

To test reassemblers with REASSESSOR, one needs to have a set of triples (α_c, β, α_r) that can reflect various code and data patterns. Thus, we create our own benchmark with various combinations of compilers, linkers, target ISAs, and compiler options. Our benchmark is created by compiling three source packages totaling 153 executable programs as follows.

- GNU coreutils (v8.30): 107 executable programs.
- GNU binutils (v2.31.1): 15 executable programs.
- SPEC CPU 2006 (v1.1): 31 executable programs.

We consider all possible combinations of the following configurations in order to produce assembly files and binaries with diverse assembly expression patterns.

- ISA: x86 and x86-64 (= 2)
- Compilers: GCC v7.5.0 and Clang v12.0 (= 2)
- Linkers: GNU ld v2.30 and GNU gold v1.15 (= 2)
- PIE/non-PIE: produce a PIE or a non-PIE (= 2)
- Optimization: O0, O1, O2, O3, Os, and Ofast (= 6)

For each executable program, we can generate 96 (= $2 \times 2 \times 2 \times 2 \times 6$) different binaries, which sums up to 14,688 binaries (= 96×153) in total. We compiled all these programs with the `-save-temps` option in order to obtain assembly files during compilation. Whenever we detect overwritten files with `loggedfs` (as discussed in §3.2), we manually modified `Makefiles` to preserve all the assembly files. We also enabled the `-g` option to produce binaries with debugging information. For each binary, we made a stripped copy by running the `strip` command. Hence, our benchmark includes a total of 14,688 not-stripped binaries and 14,688 stripped binaries.

5 Evaluation

We now evaluate existing reassemblers with REASSESSOR to identify potential reassembly challenges and their implication. In particular, we address the following research questions.

RQ1. What are the characteristics of relocatable expressions in real-world binaries? Are there any reassembly techniques that can suffer due to such characteristics? (§5.2)

RQ2. Can the current state-of-the-art reassemblers produce correct assembly files? How accurate are they? (§5.3)

RQ3. How do the symbolization errors found by REASSESSOR look? Can we get useful insights from them? (§5.4)

RQ4. Can REASSESSOR improve an existing state-of-the-art reassembler? (§5.5)

5.1 Experimental Setup

With REASSESSOR, we tested the following three state-of-the-art reassemblers: Ramblr (commit `64d1049`, Apr. 2022), RetroWrite (commit `613562`, Apr. 2022), and Ddisasm v1.5.3 (docker image digests `a803c9`, Apr. 2022). We first ran each tool with the binaries in our benchmark (§4), and collected reassembled assembly files. Next, we constructed a list of triples (α_c, β, α_r) with the generated assembly files, and ran REASSESSOR on each of the triples to discover errors in the reassemblers. Note that each tool supports different sets of binaries: Ramblr only works with non-PIE binaries and RetroWrite only works with x86-64 PIE binaries. Thus, we used only a subset of the binaries for those tools: 7,344 binaries and 3,672 for Ramblr and RetroWrite, respectively. We also provided non-stripped binaries as input to RetroWrite because it requires debugging information to operate.

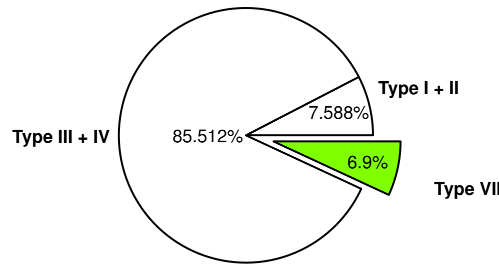


Figure 6: Proportion of each relocatable expression type for x86-64 PIE assembly files.

5.2 Statistics about Relocatable Expressions

With our custom assembly parser (§3.3.1), we examined every relocatable expression of the assembly files in our benchmark in order to understand their statistical characteristics. In particular, we answer the following questions: (1) How precise can code pointer heuristics be? (2) Do x86-64 PIE binaries have any hard-to-recover relocatable expressions? (3) How much proportion of composite relocatable expressions are there in our benchmark?

5.2.1 Reflection on Code Pointer Heuristics

Existing code pointer heuristics, such as the one used by Uroboros [89], assume that code pointers can only point to a function entry point. We used REASSESSOR to analyze all the relocatable expressions found in our benchmark to check if there is a code pointer that refers to a location other than a function entry point. As a result, we found 394 such expressions (excluding jump table entries) from 0.65% of the binaries in our benchmark. We further analyzed those assembly files to understand their uses, and found that they were mainly due to `goto` statements used in the SPEC benchmark. Thus, we conclude that existing code pointer heuristics do *not* work well with ill-structured programs with many `goto`s.

5.2.2 Breaking the Myth of x86-64 PIE Reassembly

Recall that recent reassemblers, such as RetroWrite [26], Egalito [95], and LLR [64], focus on x86-64 PIEs due to the easiness of identifying must-symbolize targets. In case there is an instruction that uses absolute addressing, the compiler will make a relocation entry in the resulting binary so that the reference can always be relocated at link time. For these reasons, some researchers have believed that x86-64 PIE reassembly can be sound without precise CFG recovery. But is this true? Are there any relocatable expressions that cannot be identified by PC-relative instructions or with the relocation table?

We answer this by analyzing the x86-64 PIE binaries (3,672 binaries in total) in our benchmark and all the corresponding compiler-generated assembly files. Specifically, we examined

```c
int output=0;
const int bar[]={-0x180, -0x190, -0x1a0, -0x1b0};
void foo(unsigned int input) {
  int *p = (int *)bar - 3;
  switch(input){
    case 0:  output = bar[0]; break;
    case 1:  output = bar[1]; break;
    case 2:  output = bar[2]; break;
    case 3:  output = bar[3]; break;
    default:
        if(input < 7) output = p[input]; break;
  }
  printf("In:%x, Out:%x\n", input, output);
}
```

(a) Source code in C.

```
.section   .text
foo:
    ;...
  lea  rax, [rip+.LJTI0_0]
    ;...
  add  rdx, rax
  jmp  rdx
    ;...
.section   .rodata
.LJTI0_0:
    .long   .LBB0_1-.LJTI0_0
    .long   .LBB0_2-.LJTI0_0
    .long   .LBB0_3-.LJTI0_0
    .long   .LBB0_4-.LJTI0_0
bar:
    .long   0xfffffe80 ; -0x180
    .long   0xfffffe70 ; -0x190
    .long   0xfffffe60 ; -0x1a0
    .long   0xfffffe50 ; -0x1b0
```

(b) x86-64 assembly (α_c).

```
.section   .text
foo:
    ;...
0x69c: lea  rax, [rip+0x18d] ; 0x830
    ;...
0x6ab: add  rdx, rax
0x6ae: jmp  rdx
    ;...
.section   .rodata
; This part corresponds to .LJTI0_0
0x830:   80 fe ff ff
0x834:   91 fe ff ff
0x838:   a2 fe ff ff
0x83c:   b3 fe ff ff
; This part corresponds to bar
0x840:   80 fe ff ff
0x844:   70 fe ff ff
0x848:   60 fe ff ff
0x84c:   50 fe ff ff
```

(c) Disassembled β.

Figure 7: Example illustrating the problem of label-relative relocatable expressions in x86-64 PIEs.

every relocatable expression in the assembly files to measure the proportion of each relocatable expression type as illustrated in Figure 6. As expected, most relocatable expressions in x86-64 PIEs are used for PC-relative addresses (Type III and Type IV), and none of the expressions is used for GOT-relative addresses (no Type V nor Type VI).

More importantly, though, we found that 6.9% of x86-64 PIEs use label-relative (Type VII) relocatable expressions, and all of them are located in a data section representing a jump table entry. This implies that *precise* CFG recovery is indeed a key requirement for reassembly even for x86-64 PIEs because one cannot recover the correct expressions without precise CFGs.

To understand why CFG recovery matters, let us consider a toy example in Figure 7 we created. Figure 7b and Figure 7c respectively show α_c and β obtained by compiling the source code with Clang to get a x86-64 PIE binary. Note there is a jump table at `.LJTI0_0` for the switch statement where each entry is in the form of "$\mathtt{label}_1 - \mathtt{label}_2$", i.e., Type VII. One may analyze the `lea` instruction as well as the following `jmp` instruction to realize that the data value at `0x830` is the start address of the jump table. However, the main challenge is to figure out where the jump table ends: Knowing the

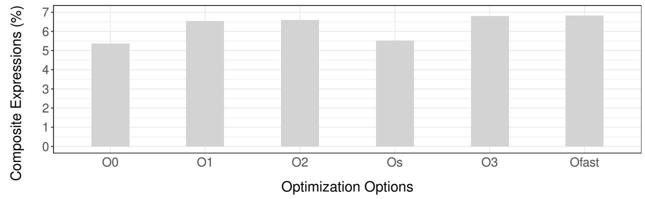


Figure 8: Proportion of composite relocatable expressions over different compiler optimization options.

precise jump table bounds implies complete CFG recovery. In this example, the global array (`bar`) immediately follows the jump table, and all the reassemblers that we tested failed to identify the correct jump table boundary, causing it to create a malfunctioning binary.

This result highlights the importance of CFG recovery even for x86-64 PIEs. Moreover, RetroWrite and Ddisasm had **E7** symbolization errors for 3.76% of the x86-64 PIE binaries in our benchmark. Thus, we conclude that precise CFG recovery is a necessary condition for sound reassembly of x86-64 PIEs.

5.2.3 Significance of Composite Expressions

Recall from §2.2, recovering composite relocatable expressions is challenging as we cannot identify the original reference unless we understand the entire program semantics. Indeed, identifying the origin of a pointer reduces to the traditional variable recovery problem [4,49,73]. Thus, it is natural to ask how many of the relocatable expressions are composite, and what is their significance.

We answer this question by measuring the proportion of composite and atomic relocatable expressions in our benchmark. First, there are a total of 266,879,967 relocatable expressions in our benchmark, and 6.28 % of them are indeed composite. Furthermore, 97.4% of the binaries in our benchmark contain at least one composite relocatable expression. Unfortunately, correctly symbolizing composite relocatable expressions is difficult: only 34.6% of the expressions were correctly symbolized.

Figure 8 describes the proportion of composite relocatable expressions for different sets of assembly files compiled with different compiler optimization options. It has turned out that we get more composite relocatable expressions as we apply more aggressive optimizations. The `Ofast` option, which is the most aggressive one, produced the most number of composite expressions (6.83%). Thus, handling composite relocatable expressions becomes more difficult when dealing with highly optimized binaries.

The problem can only become worse when the symbolization target, i.e., a displacement or an immediate in the binary, does *not* fall into a predefined memory region as indicated by [88]. We found that 1.82% of the binaries in our benchmark have at least one composite expression pointing outside of valid memory ranges. We further discuss in §5.4.3 why

Table 3: Reassembly success rates for different binary sets.

		Ramblr		RetroWrite		Ddisasm	
		Ran	Comp.*	Ran	Comp.	Ran	Comp.
GCC	coreutils	100%	95.1%	100%	100%	100%	99.4%
	binutils	97.5%	64.7%	100%	56.7%	95.0%	84.2%
	SPEC	71.6%	44.8%	96.8%	90.1%	99.2%	86.8%
Clang	coreutils	100%	99.2%	100%	99.1%	100%	98.3%
	binutils	97.5%	82.2%	100%	100%	96.5%	77.2%
	SPEC	73.7%	45.6%	93.5%	87.1%	97.5%	83.5%
Total succ. rate		94.2%	84.3%	99.3%	95.2%	99.2%	94.3%
Total succ. bins.		6,921	6,191	3,648	3,497	14,575	13,850
Total tried bins.		7,344	7,344	3,672	3,672	14,688	14,688

* Comp. means the produced assembly file compiled successfully.

existing heuristics suggested by Ddisasm and Ramblr are not enough to handle such cases.

5.3 Reassembly Errors

We now analyze the reassembly errors that REASSESSOR found from the three state-of-the-art reassemblers. While running our experiments, we found that not every binary in our benchmark is reassemblable by the reassemblers, and not every reassembled assembly file can be compiled. Table 3 summarizes the results. The "Ran" columns show the success rates of each reassembler execution, and the "Comp." columns show the success rates of each compilation attempt.

First, Ramblr, RetroWrite, and Ddisasm were able to produce an assembly file for 94.2%, 99.3%, and 99.2% of the binaries, respectively. The tools did not produce assembly files due to various runtime errors. Among the generated assembly files, 91.6% of them were compilable. Even for those files that did not compile, we were able to analyze their reassembly errors using our error-resilient parser described in §3.3.1. These results show that reassembly is still not a mature field and there is plenty of room for improvement.

5.3.1 Symbolization Errors

For all the assembly files generated by each tool, we ran REASSESSOR to identify symbolization errors. The second row of Table 4 respectively shows the numbers of reassembled binaries and the numbers of successfully reassembled binaries. The success rate was considerably low, which means that those tools had at least one symbolization error for most of each binary. Although we did not verify the reachability of those errors, this result indicates that the symbolization challenge is still largely unsolved.

The third row of Table 4 presents the numbers of symbolization errors found for each error type. Ramblr does not have **E5**–**E7** errors—marked with a dash—because it only handles non-PIE binaries while Type V–VII relocatable expressions

Table 4: Numbers of reassembly errors REASSESSOR found for each tool.

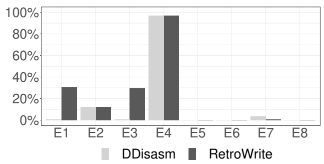
			Ramblr	RetroWrite	Ddisasm
# of Bins Reassembled			6,921	3,648	14,575
# of Bins Succeeded			200	110	221
Symbolization Errors	E1	# of TPs	28,395,297	4,137,122	41,770,473
		# of FNs	94,005	491,294	3,815,817
		# of FPs	46,144	0	54
	E2	# of TPs	52	44,976	192,764
		# of FNs	423	774	2,707,280
		# of FPs	3,879,115	43,920	2,685,997
	E3	# of TPs	64,326,100	53,917,919	177,186,331
		# of FNs	371	76	3,318,312
		# of FPs	29	52,370	33
	E4	# of TPs	4	3,614	4,735
		# of FNs	0	0	2,415,954
		# of FPs	1,405,352	2,503,910	2,283,903
	E5	# of TPs	*-	-	8,102,765
		# of FNs	-	-	3,464,715
		# of FPs	-	-	104
	E6	# of TPs	-	-	70
		# of FNs	-	-	58,846
		# of FPs	-	-	833,510
	E7	# of TPs	-	4,576,136	5,195,204
		# of FNs	-	280	128,954
		# of FPs	-	0	126
	E8	# of FPs	705,318	0	527,340
Disasm Errors		# of TPs	386,625,782	264,877,436	1,078,771,523
		# of FNs	4,235	0	1,524
		# of FPs	2,442	0	317

* The dash (-) means that the tool does not support corresponding binaries.

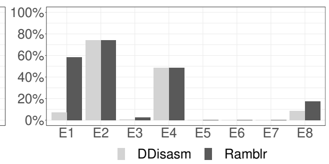
are only found in PIE binaries. RetroWrite does not have **E5**–**E6** errors because it only supports x86-64 PIE binaries while Type V and type VI relocatable expressions are only found in x86 PIE binaries. We observe that none of the reassemblers is free from symbolization errors. As we will discuss in §5.4, we were able to discover various code and data patterns that previous reassemblers do *not* handle.

It is important to note that the numbers in Table 4 indicate the numbers of symbolization errors found by reassembling binaries with each tool in our benchmark, but not the numbers of errors of each tool. That is, one may significantly reduce the numbers by fixing a heuristic or handling a specific error case. We indeed show that enhancing the current state-of-the-art tool is feasible by carefully analyzing the results (§5.5).

Since the reassemblers we tested support different sets of binaries in our benchmark, we used two different subsets of our benchmark to fairly compare the relative ability of those tools in terms of symbolization accuracy. Figure 9 illustrates two experimental results: Figure 9a compares Ddisasm and RetroWrite on x86-64 PIE binaries, and Figure 9b compares Ddisasm and Ramblr on non-PIE binaries.



(a) x86-64 PIEs.



(b) Non-PIEs.

Figure 9: Percentage of reassembled binaries that returned at least one symbolization error for each error type.

Overall, all the tools had similar performance except for **E1**, **E3**, and **E8**. RetroWrite and Ramblr had significantly more **E1** and **E3** errors compared to Ddisasm because they did not correctly handle data sections. For example, we found that RetroWrite does not handle relocation entries for read-only global variables. By not symbolizing such global variables, RetroWrite produces both **E1** and **E3** errors. Ramblr had more **E8** errors compared to Ddisasm because it aggressively symbolizes numbers, e.g., it symbolizes unaligned data [88]. To sum up, Ddisasm shows the least number of error cases compared to the other tools. Regardless, there is still ample room for improvement in the field.

5.3.2 Disassembly Errors

Recall from §3.5, REASSESSOR can also find disassembly errors during the reassembly process. It is not surprising to observe disassembly errors from existing reassemblers because disassembling binaries is challenging by itself [2, 10, 43, 62, 74]. The bottom part of Table 4 shows the number of disassembly errors found from each reassembler. Note that RetroWrite leverages debugging information to disassemble binaries, so there is no disassembly error. The other tools leverage various techniques to improve the accuracy of disassembly, but our evaluation shows that they still suffer from disassembly errors in terms of both FPs and FNs.

5.4 Dissecting Reassembly Errors

We further analyzed reassembly error cases REASSESSOR found to extract useful insights. In particular, we analyzed common patterns found in our bug database and manually analyzed several of those patterns to discover interesting ones. This section presents our findings as summarized below.

- There are previously unknown FN patterns. (§5.4.1)
- There are previously unknown FP patterns. (§5.4.2)
- Data addresses can vary with different linkers. (§5.4.3)

5.4.1 False Negatives

Previous work showed that false negative errors are mostly due to composite relocatable expressions (e.g., §6.2 of [32]), but how often can we find false negatives on atomic relocatable expressions? To answer this question, we analyzed all the error types with atomic relocatable expressions (**E1**, **E3**, and **E5**).

Surprisingly, we found numerous FNs with atomic relocatable expressions in 34.1% of the reassembled assembly files in our benchmark. For example, there is an instruction "`lea ecx, [ebx + L60c7@GOTOFF]`" from the assembly file generated for `mkdir` of coreutils. This assembly line causes a FN error for Ddisasm, because the displacement is a GOT-based offset, and Ddisasm failed to correctly analyze it.

5.4.2 False Positives

Previous research focuses on identifying and symbolizing composite relocatable expressions, but are there any cases where an atomic relocatable expression is falsely regarded as a composite expression, thereby causing a FP? For example, can the base pointer reattribution technique proposed by Ramblr [88] cause any FPs?

We found that such FPs are prevalent in practice: 5.7% of the reassembled assembly files in our benchmark had such an error. As an example, given the instruction "`lea r12, [rip+L14ef60]`" found in `strings` of coreutils, RetroWrite symbolized the displacement as "`L1110e0+0x3de80`".

We also found that a symbolization error can be cascaded to lead to another symbolization error. For example, the immediate value in "`mov edx, L4ec6fa`" is falsely symbolized by Ddisasm as "`mov edx, L4ec6f8+2`" because there exists an erroneous symbol at `0x4ec6f8` referring to a `quad` data value. This example signifies the complexity of symbolization errors found in real-world binaries.

Furthermore, we observed FP cases where symbolized labels (in α_r) have the same form as in the original (in α_c), while only the label values are misidentified. As an example, §A.13 presents a case where the labels in α_r and α_c do *not* match, while the reassembler correctly analyzed it as a Type VII relocatable expression. We found such cases in 1.9% of the reassembled assembly files.

5.4.3 Varying Data Addresses

During the course of our study, we found that linkers can also affect the shape of symbolization errors. Figure 10 describes an error case we found from two different binaries compiled with the same compiler, but with two different linkers: β^1 from gold and β^2 from ld. Note that the compiler-generated assembly file (α_c) has a composite relocatable expression `argname + 0xa0`. The resulting two binaries, even though they are from the same assembly file, have different memory layouts. As a result, the memory operand of the `lea` instruction can be symbolized in totally different ways for each binary. When we reassemble those two binaries with Ddisasm, the `lea` instruction of α_r^1 points to a data section, whereas the `lea` instruction

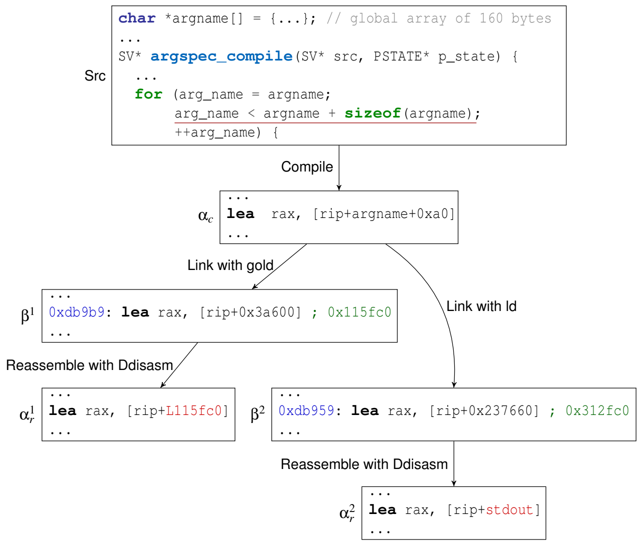


Figure 10: Error case presenting the importance of recovering composite expressions.

of α_r^2 refers to a symbol `stdout` in the `.bss` section.

This example highlights the fact that a linker can largely change the memory layout of the resulting binary, and likewise impact the reassembly performance. Therefore, it is crucial for reassemblers to employ memory-layout-agnostic techniques and heuristics.

5.5 Enhancement to Existing Reassemblers

Now that we have found plentiful symbolization errors and several previously unknown FN/FP patterns, we further verify our insights by considering ways to enhance the current state of the art. First, we created a patch for RetroWrite to resolve **E7**. Second, we analyzed how a known heuristic employed by Uroboros could help improve the performance of other tools.

5.5.1 Patching RetroWrite

Recall that **E7** errors are due to incorrectly recovered jump table entries. RetroWrite employs a pattern-based heuristic to symbolize jump table entries where only those entries that fall within a boundary of the corresponding function are considered valid. We found there are exceptional cases where a dummy (unreachable) jump table entry points to the end of a function. Such an entry will never be referenced, but it points to an address beyond the function boundary. Therefore, RetroWrite falsely computes the boundary of the jump table, and thus, misses out several jump table entries including reachable ones.

We created a patch as well as a pull request[2] that explicitly handles such unreachable entries. We compared the numbers

[2] https://github.com/HexHive/retrowrite/pull/36

Table 5: Comparison of symbolization errors before and after applying our patch.

		RetroWrite	RetroWrite (patched)
# of Bins Reassembled		3,648	3,648
E7	**# of TPs**	4,576,136	4,573,412
	# of FPs	0	0
	# of FNs	280	4

of **E7** errors before and after applying the patch. As a result, we were able to reduce 98.6% of **E7** errors as Table 5 indicates. This result highlights that our study can directly benefit the current state-of-the-art reassemblers. We leave it as future work to further improve the existing reassembly tools by considering other types of symbolization errors.

5.5.2 Data Section Heuristic

Uroboros [89] mitigates symbolization errors for non-PIE binaries by fixing the layout of data sections. That is, it always assumes that data sections have the same (fixed) memory addresses both before and after reassembling the binary. Although this technique fundamentally limits the ability of reassemblers by preventing data instrumentation, it allows robust code instrumentation without having to distinguish between number literals and pointers.

Can this heuristic be adopted to the tools that we tested to mitigate the symbolization challenge? To answer this question, we measured an empirical lower bound of the number of reparable symbolization errors when preventing data instrumentation. We chose this method because those tools do not support fixing data layouts of reassembled binaries as in Uroboros. Specifically, we discounted symbolization errors that satisfy the following conditions: (1) the error is a false positive where two relocatable expressions $e_c \in \alpha_c$ and $e_r \in \alpha_r$ mismatch; (2) the corresponding instructions have the same opcode and operands except for the relocatable expressions e_c and e_r; (3) e_c and e_r, although syntactically different, evaluate to the same address; and (4) both e_c and e_r have a single label and the labels belong to the same section. With the above criteria, we were able to reduce at least 43.25% of the symbolization errors from our benchmark. Thus, we believe fixing the layout of data sections can be a practical heuristic for reassembly especially when data instrumentation is *not* required.

6 Discussion and Future Work

Reassembly should be in accord with the development of CFG recovery techniques. Although recent research on x86-64 PIEs shows its potential, our study in §5.2.2 reveals that sound reassembly on x86-64 PIEs also requires precise CFG recovery.

Reassembly should evolve with variable recovery techniques. Recall from §5.2.3, composite relocatable expressions are widely used in real-world binaries, and previous research suggests various heuristics to handle it. However, our study in §5.4.3 shows that those heuristics suffer when the data layout changes. This can be handled by fixing the data layout as in Egalito [95], but it requires full control over the linker and the code emission processes. To leverage existing compiler tool-chains, one needs to recover variables used in composite relocatable expressions. Thus, combining existing variable recovery techniques with reassembly is an interesting direction for future work.

We need to support IR-based reassembler/recompiler. Currently, REASSESSOR only supports disassembly-based reassemblers, but not IR-based reassemblers such as Egalito. To support such a system, one needs to have a translator from IR to disassembly, and it can be promising future work.

7 Related Work

Reassembly is a recent branch of static binary rewriting, which is a technique to modify existing executables while seamlessly injecting instrumentation into them. Due to its unique capability to modify binaries without source code, it has been widely studied for diverse purposes, such as performance optimization [53, 59, 75, 86], binary hardening [18–20, 29, 41, 46, 47, 61, 62, 64, 84, 87, 92, 98, 101, 102], and binary code reuse [12, 24, 45, 97]. For a complete review of binary rewriting, we refer to the recent survey [94].

One of the key challenges to static binary rewriting is how to statically identify the cross-references in the target binary and update those references once instrumentation has been added. Since the references in the binary will be shifted relative to the instrumentation injected into the code, all cross-references in the binary will need to be recalculated. The problem, however, is that these references are *not* immediately clear as they are computed at runtime making static binary rewriting generally infeasible. Despite this challenge, static binary rewriting has gained popularity due to the lower overhead it incurs compared to other dynamic instrumentation techniques [6, 11, 50, 55, 56]. There are four ways that this challenge can be approached.

Compiler-assisted Static Rewriters. One method to circumvent the challenge of rewriting binaries is to utilize the assistance of compilers and debugging symbols. For example, ATOM [30], Plto [75], Vulcan [28], Diablo [86], Pebil [48], CCR [46], and Bolt [59] are in this category. There are several binary hardening [29, 40], monitoring [68], profiling [69, 82], and optimization [38, 81, 96] solutions built on top of these tools. However, none of these tools handles stripped binaries.

Patch-based Static Rewriters. Some rewriters tackle the challenge by preserving the layout of the original binary while patching only a part of the overall code. Since the layout is preserved, no changes are needed to fix references. Instead, the target instruction is replaced with a small trampoline which will redirect the flow to the instrumented code. This approach is also referred to as a trampoline-based approach. Detour [35], DynInst [8], Bistro [24], and E9Patch [27] are in this category, and there are many security solutions that leverage this method: code reuse [42], taint tracking [15], hardening [16–18, 62, 85], hot patching [9], monitoring [13, 80], performance profiling [3, 37], software testing [34], fuzzing [14, 51, 54], and obfuscations [70]. However, these tools do not support fine-grained instrumentation on the instruction level as the size of the target instruction can be smaller than the size of the branch instruction to patch.

Table-based Static Rewriters. Rewriters in this category make a duplicate copy of the target binary and maintain an address translation table mapping the original address to a new address in the copy. The copy is then instrumented to redirect pointers to the new address in the table whenever they are dereferenced by the original program. REINS [93], PSI [100], Multiverse [5], and μSBS [71] are in this category. Several binary hardening solutions [66, 90, 92, 99] are built on top of these tools. Although this approach does support fine-grained instrumentation, it suffers from a high time and space overhead compared to the patch-based approach due to the additional table look-ups.

Reassembly-based Static Rewriters. Recent research has introduced *reassembly*-based approaches. Reassemblers attempt to resolve the challenge by creating a relocatable representation from a binary. In this paper, we use the term "reassembly" to mean a fully static binary translation technique that does not rely on any runtime support through symbolization. Pang *et al.* [60] examined symbolization algorithms used in several binary analysis tools including reassemblers, but they did not investigate distinct types of symbolization errors, and did not provide a systematic way to discover them.

8 Conclusion

In this paper, we showed with our formal framework and an automated system that reassembly is a challenging problem even for x86-64 PIEs. Particularly, we presented REASSESSOR, the first automated system for detecting reassembly errors. Through REASSESSOR, we analyzed three existing reassemblers to find various reassembly errors with previously unknown patterns, which can be later used to improve the current state-of-the-art.

Acknowledgement

We thank Ruoyu "Fish" Wang for his constructive feedback and support for conducting our experiments. 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).

A Symbolization Error Cases.

This section showcases symbolization errors found by REASSESSOR for each error type. Labels in the assembly instructions are normalized based on the rules described in §3.4.

A.1 E1 False Positive

`mov esi, L61122a`	(α_c)
`53b803: mov rsi, 0x61122a`	(β)
`mov esi, OFFSET L611228+2`	(α_r)

This error case is found with Ddisasm when reassembling x86-64 `as-new` binary, which was compiled by GCC and ld with `-nopie` and `-O0` options. Ddisasm misidentified the atomic label `L61122a` as a composite relocatable expression.

A.2 E1 False Negative

`cmp rbx, L4e47d0`	(α_c)
`0x40b5cb: cmp rbx, 0x4e47d0 ; 0x4e47d0 is in .bss`	(β)
`cmp rbx, 0x4e47d0`	(α_r)

This error case is found with Ddisasm when reassembling x86-64 `400.perlbench` binary, which was compiled by GCC and ld with `-nopie` and `-Os` options. Ddisasm failed to identify the absolute address `0x4e47d0` as a symbolization target even though the address falls into the `.bss` section.

A.3 E2 False Positive

`mov eax, DWORD PTR [0x24+L8056300]`	(α_c)
`804cdf3: mov eax, DWORD PTR [0x8056324]`	(β)
`mov eax, DWORD PTR [L8056324]`	(α_r)

This error case is found with Ramblr when reassembling x86 `pinky` binary, which was compiled by GCC and gold with `-nopie` and `-Ofast` options. Ramblr failed to identify the composite relocatable expression `0x24+L8056300` and created a false relocation expression `L8056324`.

A.4 E2 False Negative

`movabs rax, L9f7520+0xffffffff`	(α_c)
`0x4971c7: movabs rax, 0x1009f751f`	(β)
`movabs rax, 0x1009f751f`	(α_r)

This error case is found with Ddisasm when reassembling x86-64 `403.gcc` binary, which was compiled by GCC and ld with `-nopie` and `-O1` options. Ddisasm failed to identify the relocatable expression `L9f7520+0xffffffff` and classified `0x1009f751f` as an immediate since the value points outside the `.bss` section.

A.5 E3 False Positive

`lea rcx, QWORD PTR [rip+Lbc60]`	(α_c)
`23c2: lea rcx, QWORD PTR [rip+0x9897] ; 0xbc60`	(β)
`lea rcx, QWORD PTR [rip+0x3e60+L7e00]`	(α_r)

Thie error case is found with RetroWrite when reassembling x86-64 `mktemp` binary, which was compiled by GCC and gold with `-pie` and `-O0` options. RetroWrite failed to identify the relocatable expression `Lbc60` and created a false relocatable expression because it was not able to create a symbol at `0xbc60`.

A.6 E3 False Negative

`lea rsi, QWORD PTR [rip+La6db6]`	(α_c)
`a6daa: lea rsi, QWORD PTR [rip+5] ; 0xa6db6`	(β)
`lea rsi, QWORD PTR [rip+5]`	(α_r)

This error case is found with RetroWrite when reassembling x86 `size` binary, which was compiled by Clang and gold with `-pie` and `-Os` options. RetroWrite failed to identify the relative address `0xa6db6` as a symbolization target.

A.7 E4 False Positive

`mov r13d, DWORD PTR [rip+0x24efc+L92aa60]`	(α_c)
`0x409586: mov r13d, [rip+0x5463cf] ; 0x94f95c in .bss`	(β)
`mov r13d, [rip+L94f95c]`	(α_r)

This error case is found with Ddisasm when reassembling x86-64 `445.gobmk` binary, which was compiled by GCC and gold with `-nopie` and `-Ofast` options. Ddisasm failed to identify the relocatable expression `0x24efc+L92aa60` and created a false label at a different data area.

A.8 E4 False Negative

`lea r12, [rip-0x22d00+L34140]`	(α_c)
`c26b: lea r12, [rip+0x51ce] ; 0x11440 in .text`	(β)
`lea r12, [rip+0x51ce]`	(α_r)

This error case is found with Ddisasm when reassembling x86-64 `434.zeusmp` binary, which was compiled by Clang and gold with `-pie` and `-Os` options. Ddisasm failed to identify the relocatable expression `-0x22d00+L34140`, which falls into the `.text` section.

A.9 E5 False Positive

`.long L95eb8@GOTOFF`	(α_c)
`c5fe4: c4 5e f9 ff`	(β)
`.long Le4b5-L785f1`	(α_r)

This error case is found with Ddisasm when reassembling x86 `nm-new` binary, which was compiled by GCC and gold with `-pie` and `-O2` options. Ddisasm failed to identify the relocatable expression `L95eb8@GOTOFF` and created a label-relative offset `Le4b5-L785f1` at `0x1dfe4`.

A.10 E5 False Negative

`lea eax, [ebx+L194bc@GOTOFF]`	(α_c)
`0x120ce: lea eax, [ebx-0x8b44] ;ebx holds .got addr.`	(β)
`lea eax, [ebx-0x8b44]`	(α_r)

This error case is found with Ddisasm when reassembling x86 `ls` binary, which was compiled by GCC and ld with `-pie` and `-O1` options. Ddisasm failed to identify the relocatable expression `-0x8b44` as a symbolization target because it was not able to realize that the `ebx` register holds the GOT address, 0x22000. Hence, `ebx-0x8b44` refers to the address `0x194bc` (`L194bc`), which falls into the `.rodata` section.

A.11 E6 False Positive

`push DWORD PTR [ebx+0x2c+L1e2e0@GOTOFF]`	(α_c)
`c63d: push DWORD PTR [ebx+0x30c] ; 0x1e30c`	(β)
`push DWORD PTR [ebx+L1e30c@GOTOFF]`	(α_r)

This error case is found with Ddisasm when reassembling x86 `touch` binary, which was compiled by GCC and ld with `-pie` and `-O3` options. Ddisasm failed to identify the relocatable expression `0x2c+L1e2e0@GOTOFF` and created an atomic label `L1e30c@GOTOFF`.

A.12 E6 False Negative

`lea eax, DWORD PTR [ebx+4+L171e0@GOTOFF]`	(α_c)
`1c7c: lea eax, DWORD PTR [ebx+0x1e4] ;0x171e4`	(β)
`lea eax, DWORD PTR [ebx+0x1e4]`	(α_r)

This error is found with Ddisasm when reassembling x86 `stty` binary, which was compiled by GCC and ld with `-pie` and `-O3` options. Ddisasm failed to identify the relocatable expression `4+L171e0@GOTOFF` because it was not able to create a symbol at `0x171e4`.

A.13 E7 False Positive

`L3c75cc:`
`.long L2ca3f0-L3c75cc`
`.long L2ca758-L3c75cc`
	(α_c)

`0x3c75cc: 24 2e f0 ff` `0x3c75d0: 8c 31 f0 ff`	(β)

`L3c75cc:`
`.long L2c8204-L3c53e0 ; E7FP`
`.long L2ca758-L3c75cc`
	(α_r)

This error case is found with Ddisasm when reassembling x86-64 `403.gcc` binary, which was compiled by GCC and gold with `-pie` and `-O3` options. Ddisasm symbolized the relocatable expression `L2ca3f0-L3c75cc` to `L2c8204-L3c53e0`, causing a false positive.

A.14 E7 False Negative

`L5b40c:`
`.long L251df-L5b40c`
`.long L26b94-L5b40c`
	(α_c)

`0x5b40c: d3 9d fc ff` `0x5b410: 88 b7 fc ff`	(β)

`L5b40c:`
`.long L251df-L5b40c`
`L5b410:`
`.byte 0x88`
	(α_r)

This error case is found with RetroWrite when reassembling x86-64 `readelf` binary, which was compiled by Clang and ld with `-pie` and `-O1` options. RetroWrite failed to identify the relocatable expressions located at `0x5b410`.

A.15 E8 False Positive

`add [ebp-0xa0], 0x20000000`	(α_c)
`0x805be86: add [ebp-0xa0], 0x20000000`	(β)
`add [ebp-0xa0], L20000000`	(α_r)

This error case is found with Ramblr when reassembling x86 `434.zeusmp` binary, which was compiled by Clang and gold with `-no-pie` and `-Os` options. Ramblr falsely symbolized the immediate `0x20000000` since the value falls into the `.bss` section.

References

[1] 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.

[2] Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In *Proceedings of the USENIX Security Symposium*, pages 583–600, 2016.

[3] Mahwish Arif, Ruoyu Zhou, Hsi-Ming Ho, and Timothy M. Jones. Cinnamon: A domain-specific language for binary profiling and mon-

itoring. In *IEEE/ACM International Symposium on Code Generation and Optimization*, pages 103–114, 2021.

[4] Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 executables. In *Proceedings of the International Conference on Compiler Construction*, pages 5–23, 2004.

[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] Andrew R. Bernat and Barton P. Miller. Structured binary editing with a cfg transformation algebra. In *Working Conference on Reverse Engineering*, pages 9–18, 2012.

[10] Guillaume Bonfante, Jose Fernandez, Jean-Yves Marion, Benjamin Rouxel, Fabrice Sabatier, and Aurélien Thierry. CoDisasm: Medium scale concatic disassembly of self-modifying binaries with overlapping instructions. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 745–756, 2015.

[11] 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.

[12] Juan Caballero, Noah M Johnson, Stephen McCamant, and Dawn Song. Binary code extraction and interface identification for security applications. In *Proceedings of the Network and Distributed System Security Symposium*, 2010.

[13] Wu chang Feng, Ed Kaiser, and Travis Schluessler. Stealth measurements for cheat detection in on-line games. In *Proceedings of the ACM SIGCOMM Workshop on Network and System Support for Games*, pages 15–20, 2008.

[14] Hongxu Chen, Yuekang Li, Bihuan Chen, Yinxing Xue, and Yang Liu. Fot: a versatile, configurable, extensible fuzzing framework. In *Proceedings of the International Symposium on Foundations of Software Engineering*, pages 867–870, 2018.

[15] Sanchuan Chen, Zhiqiang Lin, and Yinqian Zhang. SelectiveTaint: Efficient data flow tracking with static binary rewriting. In *Proceedings of the USENIX Security Symposium*, pages 1665–1682, 2021.

[16] Ting Chen, Yang Xu, and Xiaosong Zhang. A program manipulation middleware and its applications on system security. In *International Conference on Security and Privacy in Communication Systems*, pages 606–626, 2018.

[17] Xi Chen, Herbert Bos, and Cristiano Giuffrida. Codearmor: Virtualizing the code space to counter disclosure attacks. In *Proceedings of the IEEE European Symposium on Security and Privacy*, pages 514–529, 2017.

[18] Xi Chen, Asia Slowinska, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. Stackarmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries. In *Proceedings of the Network and Distributed System Security Symposium*, 2015.

[19] Yaohui Chen, Dongli Zhang, Ruowen Wang, Rui Qiao, Ahmed M Azab, Long Lu, Hayawardh Vijayakumar, and Wenbo Shen. NORAX: Enabling execute-only memory for COTS binaries on aarch64. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 304–319, 2017.

[20] Yue Chen, Zhi Wang, David Whalley, and Long Lu. Remix: On-demand live randomization. In *Proceedings of the ACM Conference on Data and Application Security and Privacy*, pages 50–61, 2016.

[21] Anton. Chernoff, Mark Herdeg, Ray Hookway, Chris Reeve, Norman Rubin, Tony Tye, S. Bharadwaj Yadavalli, and John Yates. FX!32 a profile-directed binary translator. *IEEE Micro*, 18(2):56–64, 1998.

[22] Christina Cifuentes and Vishv Malhotra. Binary translation: Static, dynamic, retargetable? In *Proceedings of International Conference on Software Maintenance*, pages 340–349, 1996.

[23] Cristina Cifuentes and Mike Van Emmerik. UQBT: Adaptable binary translation at low cost. *Computer*, 33(3):60–66, 2000.

[24] 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.

[25] Artem Dinaburg and Andrew Ruef. McSema: Static translation of x86 instructions to LLVM. In *Proceedings of the Reverse Engineering and Security Conference*, 2014.

[26] 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.

[27] 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.

[28] Andrew Edwards, Amitabh Srivastava, and Hoi Vo. Vulcan: Binary transformation in a distributed environment. Technical Report MSR-TR-2001-50, Microsoft Research, 2001.

[29] 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.

[30] Alan Eustace and Amitabh Srivastava. ATOM: A flexible interface for building high performance program analysis tools. In *Proceedings of the USENIX Technical Conference*, pages 303–314, 1995.

[31] Rémi Flament. LoggedFs: Filesystem monitoring with fuse. https://github.com/rflament/loggedfs, 2016.

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

[33] William H Hawkins, Jason D Hiser, Michele Co, Anh Nguyen-Tuong, and Jack W Davidson. Zipr: Efficient static binary rewriting for security. In *Proceedings of the International Conference on Dependable Systems Networks*, pages 559–566, 2017.

[34] Yao-Wen Huang, Shih-Kun Huang, Tsung-Po Lin, and Chung-Hung Tsai. Web application security assessment by fault injection and behavior monitoring. In *Proceedings of the international conference on World Wide Web*, pages 148–159, 2003.

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

[36] Intel Corporation. Intel® 64 and ia-32 architectures software developer's manual. https://software.intel.com/en-us/articles/intel-sdm.

[37] Rebecca Isaacs and Paul Barham. Performance analysis in loosely-coupled distributed systems. In *CaberNet Radicals Workshop*, 2002.

[38] Timothy M. Jones, Sandro Bartolini, Jonas Maebe, and Dominique Chanet. Link-time optimization for power efficiency in a tagless instruction cache. In *International Symposium on Code Generation and Optimization*, pages 32–41, 2011.

[39] 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.

[40] Ronald De Keulenaer, Jonas Maebe, Koen De Bosschere, and Bjorn De Sutter. Link-time smart card code hardening. *International Journal of Information Security*, 15(2):111–130, 2016.

[41] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *Proceedings of the Annual Computer Security Applications Conference*, pages 339–348, 2006.

[42] Dohyeong Kim, William N. Sumner, Xiangyu Zhang, Dongyan Xu, and Hira Agrawal. Reuse-oriented reverse engineering of functional components from x86 binaries. In *Proceedings of the International Conference on Software Engineering*, pages 1128–1139, 2014.

[43] 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 Networks*, pages 559–566, 2022.

[44] Soomin Kim, Markus Faerevaag, Minkyu Jung, Seungil Jung, DongYeop Oh, JongHyup Lee, and Sang Kil Cha. Testing intermediate representations for binary analysis. In *Proceedings of the International Conference on Automated Software Engineering*, pages 353–364, 2017.

[45] Clemens Kolbitsch, Thorsten Holz, Christopher Kruegel, and Engin Kirda. Inspector gadget: Automated extraction of proprietary gadgets from malware binaries. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 29–44, 2010.

[46] Hyungjoon Koo, Yaohui Chen, Long Lu, Vasileios P Kemerlis, and Michalis Polychronakis. Compiler-assisted code randomization. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 461–477, 2018.

[47] Hyungjoon Koo and Michalis Polychronakis. Juggling the gadgets: Binary-level code randomization using instruction displacement. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security*, pages 23–34, 2016.

[48] 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.

[49] JongHyup Lee, Thanassis Avgerinos, and David Brumley. TIE: Principled reverse engineering of types in binary programs. In *Proceedings of the Network and Distributed System Security Symposium*, pages 251–268, 2011.

[50] 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.

[51] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 2019.

[52] Cathy May. Mimic: A fast system/370 simulator. *ACM SIGPLAN Notices*, 22(7):1–13, 1987.

[53] Robert Muth, Saumya K Debray, Scott Watterson, and Koen De Bosschere. Alto: A link–time optimizer for the compaq alpha. *Software: Practice and Experience*, 31(1):67–101, 2001.

[54] Stefan Nagy and Matthew Hicks. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 787–802, 2019.

[55] 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.

[56] 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.

[57] Aleksandar Nikolic and Marc Heuse. afl-dyninst. `https://github.com/talos-vulndev/afl-dyninst`, 2016.

[58] Pádraig O'sullivan, Kapil Anand, Aparna Kotha, Matthew Smithson, Rajeev Barua, and Angelos D Keromytis. Retrofitting security in COTS software with binary rewriting. In *Proceedings of IFIP TC 11 International Information Security Conference*, pages 154–172, 2011.

[59] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. Bolt: A practical binary optimizer for data centers and beyond. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 2–14, 2019.

[60] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. SoK: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2021.

[61] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 601–615, 2012.

[62] Manish Prasad and Tzi-cker Chiueh. A binary rewriting defense against stack based buffer overflow attacks. In *Proceedings of the USENIX Annual Technical Conference*, pages 211–224, 2005.

[63] Leon Presser and John R. White. Linkers and loaders. *ACM Computing Surveys*, 4(3):149–167, sep 1972.

[64] Soumyakant Priyadarshan, Huan Nguyen, and R. Sekar. Practical fine-grained binary code randomization. In *Proceedings of the Annual Computer Security Applications Conference*, pages 401–414, 2020.

[65] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. RAZOR: A framework for post-deployment software debloating. In *Proceedings of the USENIX Security Symposium*, pages 1733–1750, 2019.

[66] Rui Qiao, Mingwei Zhang, and R. Sekar. A principled approach for rop defense. In *Proceedings of the Annual Computer Security Applications Conference*, pages 101–110, 2015.

[67] Nguyen Anh Quynh. Capstone engine. `https://github.com/aquynh/capstone`, 2013.

[68] Mohan Rajagopalan, Matti A. Hiltunen, Trevor Jim, and Richard D. Schlichting. System call monitoring using authenticated system calls. *IEEE Transactions on Dependable and Secure Computing*, 3(3):216–229, 2006.

[69] Mohan Rajagopalan, Somu Perianayagam, HaiFeng He, Gregory Andrews, and Saumya Debray. Binary rewriting of an operating system kernel. In *Proceedings of the ACM ICPS Workshop on Binary Instrumentation and Applications*, 2006.

[70] Kevin A. Roundy and Barton P. Miller. Binary-code obfuscations in prevalent packer tools. *ACM Computing Surveys*, 46(1):1–32, 2013.

[71] 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.

[72] Eric Schulte, Jonathan DiLorenzo, Westley Weimer, and Stephanie Forrest. Automated repair of binary and assembly programs for cooperating embedded devices. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 317–328, 2013.

[73] Edward J. Schwartz, JongHyup Lee, Maverick Woo, and David Brumley. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *Proceedings of the USENIX Security Symposium*, pages 353–368, 2013.

[74] Benjamin Schwarz, Saumya Debray, and Gregory Andrews. Disassembly of executable code revisited. In *Proceedings of the Working Conference on Reverse Engineering*, pages 45–54, 2002.

[75] Benjamin Schwarz, Saumya Debray, Gregory Andrews, and Matthew Legendre. Plto: A link-time optimizer for the Intel IA-32 architecture. In *Proceedings of the Workshop on Binary Translation*, 2001.

[76] Kevin Scott, Naveen Kumar, Sivakumar Velusamy, Bruce Childers, Jack W. Davidson, and Mary Lou Soffa. Retargetable and reconfigurable software dynamic translation. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 36–47, 2003.

[77] Bor-Yeh Shen, Jiunn-Yeu Chen, Wei-Chung Hsu, and Wuu Yang. LLBT: an LLVM-based static binary translator. In *Proceedings of International Conference on Compilers*, pages 51–60, 2012.

[78] Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. Binary translation. *Communications of the ACM*, 36(2):69–81, 1993.

[79] Matthew Smithson, Khaled ElWazeer, Kapil Anand, Aparna Kotha, and Rajeev Barua. Static binary rewriting without supplemental information: Overcoming the tradeoff between coverage and correctness. In *Proceedings of the Working Conference on Reverse Engineering*, pages 52–61, 2013.

[80] Elizabeth Stinson and John C. Mitchell. Web application security assessment by fault injection and behavior monitoring. In *Proceedings of the international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 89–108, 2007.

[81] Bjorn De Sutter, Ludo Van Put, Dominique Chanet, Bruno De Bus, and Koen De Bosschere. Link-time compaction and optimization of arm executables. *ACM Transactions on Embedded Computing Systems*, 6(1):1–43, 2007.

[82] Ananta Tiwari, Martin Schulz, and Laura Carrington. Predicting optimal power allocation for cpu and dram domains. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 951–959, 2015.

[83] David Ung and Cristina Cifuentes. Machine-adaptable dynamic binary translation. In *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, pages 41–51, 2000.

[84] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, efficient in-process isolation with protection keys (MPK). In *Proceedings of the USENIX Security Symposium*, pages 1221–1238, 2019.

[85] Victor van der Veen, Enes Goktas, Moritz Contag, Andre Pawlowski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 934–953, 2016.

[86] Ludo Van Put, Dominique Chanet, Bruno De Bus, Bjorn De Sutter, and Koen De Bosschere. Diablo: A reliable, retargetable and extensible link-time rewriting framework. In *Proceedings of the IEEE International Symposium on Signal Processing and Information Technology*, pages 7–12, 2005.

[87] Minghua Wang, Heng Yin, Abhishek Vasisht Bhaskar, Purui Su, and Dengguo Feng. Binary code continent: Finer-grained control flow integrity for stripped binaries. In *Proceedings of the Annual Computer Security Applications Conference*, pages 331–340, 2015.

[88] 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.

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

[90] Wenhao Wang, Xiaoyang Xu, and Kevin W. Hamlen. Object flow integrity. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pages 1909–1924, 2017.

[91] Wenwen Wang, Pen-Chung Yew, Antonia Zhai, and Stephen McCamant. A general persistent code caching framework for dynamic binary translation (dbt). In *Proceedings of the USENIX Annual Technical Conference*, pages 591–603, 2016.

[92] 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.

[93] 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.

[94] Matthias Wenzl, Georg Merzdovnik, Johanna Ullrich, and Edgar Weippl. From hack to elaborate technique—a survey on binary rewriting. *ACM Computing Surveys*, 52(3):1–37, 2019.

[95] 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.

[96] Linda S. Wilson, Craig A. Neth, and Michael J. Rickabaugh. Delivering binary object modification tools for program analysis and optimization. *Digital Technical Journal*, 8(1):18–31, 1996.

[97] 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.

[98] 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.

[99] Mingwei Zhang, Michalis Polychronakis, and R. Sekar. Protecting cots binaries from disclosure-guided code reuse attacks. In *Proceedings of the Annual Computer Security Applications Conference*, pages 128–140, 2017.

[100] 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.

[101] Mingwei Zhang, Ravi Sahita, and Daiping Liu. executable-only-memory-switch (xom-switch): Hiding your code from advanced code reuse attacks in one shot. In *Proceedings of the Black Hat USA*, 2018.

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

[103] Cindy Zheng and Carol Thompson. PA-RISC to IA-64: Transparent execution, no recompilation. *Computer*, 33(3):47–52, 2000.