Reassembly is Hard:
A Reflection on Challenges and Strategies

Hyungseok Kim, KAIST and The Affiliated Institute of ETRI; Soomin Kim and Junoh Lee, KAIST; Kangkook Jee, University of Texas at Dallas; Sang Kil Cha, KAIST

https://www.usenix.org/conference/usenixsecurity23/presentation/kim-hyungseok
Reassembly is Hard: A Reflection on Challenges and Strategies

Hyungseok Kim1,2, Soomin Kim1, Junoh Lee1, Kangkook Jee3, and Sang Kil Cha4
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 REASSERSON 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) or by randomizing code layouts. It has also been well developed in other domains such as malware analysis. Table-based approaches such as PSI can be re-positioned without having to modify their source code. This technique is often used to harden legacy binaries 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. 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, 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).

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 differencing 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 (αc). For brevity, we assume that both compilers and reassemblers produce only a single assembly file αc 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 αc be an assembly file obtained from a compiler, and let β be the binary obtained by assembling αc. We denote the assembly process by Asm. That is, Asm(αc) = β. We then let αr be the assembly file obtained by reassembling β without adding any instrumentation. We use Reasm to denote the reassembly process: Reasm(β) = αr. Figure 1 illustrates the relationships between αc, αr, and β. To detect symbolization errors, we analyze the difference of the labels in αc and αr.
Normalization. To ease the comparison between αr and α, we assume that both αr and α 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, 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 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.

Table 1: Categorization of relocatable expressions.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Atomic</th>
<th>Composite</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absolute address</td>
<td>Type I</td>
<td>Type II</td>
</tr>
<tr>
<td>PC-relative address</td>
<td>Type III</td>
<td>Type IV</td>
</tr>
<tr>
<td>GOT-relative address</td>
<td>Type V</td>
<td>Type VI</td>
</tr>
<tr>
<td>Label-relative address</td>
<td>-</td>
<td>Type VII</td>
</tr>
</tbody>
</table>

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 “label1 (op) label2”, 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 $m$th instruction of $\alpha_c$ with \(\text{Code}(\alpha_c)[m].\text{disp}\).

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, Data(\(\alpha_c\))[\(m\)].value = null and Data(\(\alpha_c\))[\(m + 4\)].value = 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 $\beta$. This function makes explicit the relationship between two assembly lines respectively in $\alpha_c$ and $\alpha_r$ by referring to the address in the binary $\beta$. The red boxes in Figure 1 shows that Addr returns the address 0x1129 for both Code(\(\alpha_c\))[\(n\)] and Code(\(\alpha_r\))[\(n\)].

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

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

\[
\begin{align*}
\text{Addr}(\text{Code}(\alpha_c)[m]) &= \text{Addr}(\text{Code}(\alpha_r)[n]) \\
\land & \quad \text{Code}(\alpha_c)[m] \neq \text{Code}(\alpha_r)[n] \\
\lor & \quad \text{Data}(\alpha_c)[m] \neq \text{Data}(\alpha_r)[n]
\end{align*}
\]

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 $\beta = \text{Asm}(\alpha_c)$, while the corresponding assembly line in $\alpha_r$ has a relocatable expression.

\(^1\)A displacement is a number in a memory operand, e.g., 42 in $\text{mov} \text{ rax, [rdx + 42]}$. An immediate is a number-only operand, e.g., 42 in $\text{push} \text{ r} 4 2$. 

```
1 char msg[] = "Hi Reassembler\n";
2 void foo()
3 {
4   for(char *p = msg; p < msg + sizeof(msg); ++p)
5       putchar(*p);
6 }
```

(a) Source code in C.

```
.section .text
Disassemble of section .text:
foo:
  0x628: push r14
  0x62a: push rbx
  0x63b: push rax
  0x63c: lea rbx, [rip+msg]
  0x63d: mov eax, [rbx]
LBB0_1:
  0x63e: xor eax, eax
mov ax, byte ptr [rbx]
  0x63f: call 520
xor eax, eax
  0x644: inc rbx
call putcharPLT
  0x647: cmp rbp, r14
inc rbx
  0x64a: jmp 63a
cmp rbp, r14
  0x64c: add rsp, 0x8
jmp LBB0_1
add rsp, 0
pop rbp
  0x651: pop r14
pop r14
... ret
```

(b) x86-64 assembly code

```
.section .data
Contents of section .data
0x201000: 40 69 20 ... ; "Hi Reassembler"";
[section .bss
..section .text
Disassemble of section .text:
```
```
(c) Disassembled PIE binary code. produced by Clang.

\(\begin{align*}
\text{Addr}(\text{Code}(\alpha_c)[m]) &= \text{Addr}(\text{Code}(\alpha_r)[n]) \\
\land & \quad \text{Code}(\alpha_c)[m].\text{disp} \neq \text{null} \\
\land & \quad \text{Code}(\alpha_r)[n].\text{disp} \neq \text{null}
\end{align*}\)

\(\begin{align*}
\text{Addr}(\text{Code}(\alpha_c)[m]) &= \text{Addr}(\text{Code}(\alpha_r)[n]) \\
\land & \quad \text{Code}(\alpha_c)[m].\text{imm} \neq \text{null} \\
\land & \quad \text{Data}(\alpha_c)[m].\text{value} \neq \text{null}
\end{align*}\)

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

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

\(\begin{align*}
\text{Addr}(\text{Code}(\alpha_c)[m]) &= \text{Addr}(\text{Code}(\alpha_r)[n]) \\
\land & \quad \text{Code}(\alpha_c)[m].\text{disp} \neq \text{null} \\
\land & \quad \text{Addr}(\text{Code}(\alpha_r)[n].\text{disp} \neq \text{null}
\end{align*}\)

\(\begin{align*}
\text{Addr}(\text{Code}(\alpha_c)[m]) &= \text{Addr}(\text{Code}(\alpha_r)[n]) \\
\land & \quad \text{Code}(\alpha_c)[m].\text{imm} \neq \text{null} \\
\land & \quad \text{Data}(\alpha_c)[m].\text{value} \neq \text{null}
\end{align*}\)

\(\begin{align*}
\text{Addr}(\text{Code}(\alpha_r)[n]) &= \text{Addr}(\text{Code}(\alpha_c)[m]) \\
\land & \quad \text{Code}(\alpha_r)[n].\text{imm} \neq \text{null} \\
\land & \quad \text{Data}(\alpha_r)[n].\text{value} \neq \text{null}
\end{align*}\)

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

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

\(\begin{align*}
\text{Addr}(\text{Code}(\alpha_c)[m]) &= \text{Addr}(\text{Code}(\alpha_r)[n]) \\
\land & \quad \text{Code}(\alpha_c)[m].\text{disp} \neq \text{null} \\
\land & \quad \text{Addr}(\text{Code}(\alpha_r)[n].\text{disp} \neq \text{null}
\end{align*}\)

\(\begin{align*}
\text{Addr}(\text{Code}(\alpha_c)[m]) &= \text{Addr}(\text{Code}(\alpha_r)[n]) \\
\land & \quad \text{Code}(\alpha_c)[m].\text{imm} \neq \text{null} \\
\land & \quad \text{Data}(\alpha_c)[m].\text{value} \neq \text{null}
\end{align*}\)

\(\begin{align*}
\text{Addr}(\text{Code}(\alpha_r)[n]) &= \text{Addr}(\text{Code}(\alpha_c)[m]) \\
\land & \quad \text{Code}(\alpha_r)[n].\text{imm} \neq \text{null} \\
\land & \quad \text{Data}(\alpha_r)[n].\text{value} \neq \text{null}
\end{align*}\)
Table 2: Categorization of symbolization errors.

<table>
<thead>
<tr>
<th>ID</th>
<th>Relocatable Expression in ( \alpha_c )</th>
<th>Observable</th>
<th>32</th>
<th>64</th>
<th>PIE</th>
<th>noPIE</th>
<th>FP</th>
<th>FN</th>
<th>Ex.</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>Atomic Absolute</td>
<td>I</td>
<td>✓✓✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>FP</td>
<td>FN</td>
<td>§A.1</td>
</tr>
<tr>
<td>E2</td>
<td>Composite Absolute</td>
<td>II</td>
<td>✓✓✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>FP</td>
<td>FN</td>
<td>§A.2</td>
</tr>
<tr>
<td>E3</td>
<td>Atomic PC-rel</td>
<td>III</td>
<td>✓✓✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>FP</td>
<td>FN</td>
<td>§A.3</td>
</tr>
<tr>
<td>E4</td>
<td>Composite PC-rel</td>
<td>IV</td>
<td>✓✓✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>FP</td>
<td>FN</td>
<td>§A.4</td>
</tr>
<tr>
<td>E5</td>
<td>Atomic GOT-rel</td>
<td>V</td>
<td>✓✓✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>FP</td>
<td>FN</td>
<td>§A.5</td>
</tr>
<tr>
<td>E6</td>
<td>Composite GOT-rel</td>
<td>VI</td>
<td>✓✓✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>FP</td>
<td>FN</td>
<td>§A.6</td>
</tr>
<tr>
<td>E7</td>
<td>Composite Lab-rel</td>
<td>VII</td>
<td>✓✓✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>FP</td>
<td>FN</td>
<td>§A.7</td>
</tr>
<tr>
<td>E8</td>
<td>Constant</td>
<td>-</td>
<td>✓✓✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>FP</td>
<td>FN</td>
<td>§A.8</td>
</tr>
</tbody>
</table>

2.3 Categorization of Symbolization Errors

Recall from §2.2, a symbolization error occurs when there is a mismatch between two corresponding relocatable expressions (\( \text{reloceXpr} \)) respectively in \( \alpha_c \) and \( \alpha_r \). 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 \text{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 how we address the challenges. Finally, we discuss the soundness of our system as well as the implementation details of REASSESSOR.

3.1 Overview

At a high level, REASSESSOR takes in as input a compiler-generated assembly file (\( \alpha_c \)), a binary file (\( \beta \)), and a reassembler-generated assembly file (\( \alpha_r \)). 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 (\( \alpha_c, \beta, \alpha_r \)). The CONCAT module merges all the assembly files generated by the compiler into one. The STRIP module strips off debug symbols from the binary \( \beta \) to get a stripped binary \( \beta' \). 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 \( \beta \) 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 (\( \alpha_c, \beta, \alpha_r \)), we run ADDRMAPPER twice with two different inputs: (\( \alpha_c, \beta \)) and (\( \alpha_r, \beta \)). This way we can obtain two annotated assembly files: \( \alpha_c' \) and \( \alpha_r' \). §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: \( \alpha_c'' \) and \( \alpha_r'' \). We describe the detailed implementation in §3.4.

Finally, the DIFFER module takes in the two normalized assembly files (\( \alpha_c'' \) and \( \alpha_r'' \)) 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 \( \beta \) with assembly lines in \( \alpha \) to produce \( \alpha' \), 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 \( \alpha \) for a function in \( \beta \). We handle this challenge by comparing instruction sequences.

Specifically, we associate the address in \( \beta \) with every assembly instruction in \( \alpha \) in the following three steps. First, we enumerate every function in \( \beta \) with the help of the debugging information. Second, for each function, we find all possible functions in \( \alpha \). Third, for each function found in the previous step, we identify a matching function in \( \beta \) 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 \( \beta \), but not in \( \alpha \). Such no-op instructions have many different forms, e.g., “nop”, “nop DWORD ptr [eax+eax*1+0x0D]”, “lea esi, [esi]”, and so forth. REASSER hooks 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 \( \alpha' \).** 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 (\( \alpha' \)) 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 \( \beta \) with the debugging informa-
Figure 5: Calculating data addresses from local symbols.

### Data Addresses for \( \alpha_r \)
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 \( \alpha_r \) 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 \( \alpha' \) into another annotated assembly file \( \alpha'' \), 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 Differing
The last step of REASSESSOR is DIFFER, which compares two annotated assembly files \( \alpha''_r \) and \( \alpha''_r \) 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 differ. For every mismatch found, it examines the mismatched expression in both \( \alpha''_r \) and \( \alpha''_r \) 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 \((\alpha_r, \beta, \alpha_r)\) 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.
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 × 2 × 2 × 2 × 6) different binaries, which sums up to 14,688 binaries (= 96 × 153) in total. We compiled all these programs with the `--save-temp`s 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, β, γ) 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.

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 `gotos`.

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

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. 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...
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 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 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.
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 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 $\alpha_r$) have the same form as in the original (in $\alpha_c$), while only the label values are misidentified. As an example, §A.13 presents a case where the labels in $\alpha_r$ and $\alpha_c$ 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: $\beta^1$ from gold and $\beta^2$ from ld. Note that the compiler-generated assembly file ($\alpha_c$) has a composite relocatable expression $\text{argvname + 0xa0}$. 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 $\alpha_r^1$ points to a data section, whereas the lea instruction of $\alpha_r^2$ points to a data section.
of $\alpha_2^2$ 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 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 reassembler 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.

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

<table>
<thead>
<tr>
<th></th>
<th>RetroWrite</th>
<th>RetroWrite (patched)</th>
</tr>
</thead>
<tbody>
<tr>
<td># of Bins Reassembled</td>
<td>3,648</td>
<td>3,648</td>
</tr>
<tr>
<td># of TPs</td>
<td>4,576,136</td>
<td>4,573,412</td>
</tr>
<tr>
<td># of FPs</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td># of FNs</td>
<td>280</td>
<td>4</td>
</tr>
</tbody>
</table>

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 these 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], Plato [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

```asm
mov esi, L61122a
53b803: mov rsi, 0x61122a
mov esi, OFFSET L61122a+2
```

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

```asm
cmp rbx, L4e47d0
0x40b5cb: cmp rbx, 0x4e47d0 ; 0x4e47d0 is in .bss
cmp rbx, 0x4e47d0
```

This error case is found with Ddisasm when reassembling x86-64 400.peribench 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

```asm
mov eax, DWORD PTR [0x24+L8056300]
804cdf3: mov eax, DWORD PTR [0x8056324]
mov eax, DWORD PTR [L8056324]
```

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

```asm
movabs rax, L9f7520+0xffffffff
0x4971c7: movabs rax, 0x1009f751f
movabs rax, 0x1009f751f
```

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

```asm
lea rcx, QWORD PTR [rip+Lbc60]
lea rcx, QWORD PTR [rip+0x9897] ; 0xbc60
lea rcx, QWORD PTR [rip+0x3e60+L7e00]
```

This 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

```asm
lea rsi, QWORD PTR [rip+L6db6]
a6daa: lea rsi, QWORD PTR [rip+5] ; 0xa6db6
lea rsi, QWORD PTR [rip+5]
```

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

```asm
mov r13d, DWORD PTR [rip+0x24efc+L92aa60]
0x49f586: mov r13d, [rip+0x5463cf] ; 0x94f5c in .bss
mov r13d, [rip+L94f59c]
```

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

```asm
lea r12, [rip-0x22d00+L34140]
c268: lea r12, [rip+0x51ce] ; 0x11440 in .text
lea r12, [rip+0x51ce]
```

This error case is found with Ddisasm when reassembling
This error 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`
- `.long Le4b5-L785f1`

This error case is found with Ddisasm when reassembling x86-64 434.zeusmp binary, which was compiled by GCC and gold with -pie and -Os 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]`
- `lea eax, [ebx-0x8b44]`

This error case is found with Ddisasm when reassembling x86-64 434.zeusmp 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+0xc0+L1e2e0@GOTOFF]`
- `push DWORD PTR [ebx+0x8a90c]`

This error case is found with Ddisasm when reassembling x86-64 434.zeusmp binary, which was compiled by GCC and ld with -pie and -O3 options. Ddisasm failed to identify the relocatable expression 0xc0+L1e2e0@GOTOFF and created an atomic label L1e30c@GOTOFF.

### A.12 E6 False Negative

- `lea eax, DWORD PTR [ebx+4+L171e0@GOTOFF]`
- `lea eax, DWORD PTR [ebx+0x1e4]`

This error is found with Ddisasm when reassembling x86-64 434.zeusmp 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

- `.long L2ca3f0-L3c75cc`
- `.long L2ca758-L3c75cc`

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 L2ca8204-L3c53e0, causing a false positive.

### A.14 E7 False Negative

- `.long L251df-L5b40c`
- `.long L26b94-L5b40c`

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`
- `add [ebp-0xa0], 0x20000000`

This error case is found with Ramblr when reassembling x86-64 434.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

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


[60] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. S0K. All you ever wanted to know about x86/64 binary disassembly but were afraid to ask. In Proceedings of the IEEE Symposium on Security and Privacy, 2021.


