Adding Compilation Metadata To Binaries To Make Disassembly Decidable

Daniel Engel[1], Freek Verbeek[1], Pranav Kumar[2], and Binoy Ravindran[2]

[1] Open University, Heerlen, The Netherlands
{daniel.engel, freek.verbeek}@ou.nl
[2] Virginia Tech, Blacksburg, USA
{pranavkumar, binoy}@vt.edu

Abstract. The binary executable format is the standard method for distributing and executing software. Yet, it is also as opaque a representation of software as can be. If the binary format were augmented with metadata that provides security-relevant information, such as which data is intended by the compiler to be executable instructions, or how memory regions are expected to be bounded, that would dramatically improve the safety and maintainability of software. In this paper, we propose a binary format that is a middle ground between a stripped black-box binary and open source. We provide a tool that generates metadata capturing the compiler's intent and inserts it into the binary. This metadata enables lifting to a correct and recompilable higher-level representation and makes analysis and instrumentation more reliable. Our evaluation shows that adding metadata does not affect runtime behavior or performance. Compared to DWARF, our metadata is roughly 17% of its size. We validate correctness by compiling a comprehensive set of real-world C and C++ binaries and demonstrating that they can be lifted, instrumented, and recompiled without altering their behavior.

Keywords: Security and privacy → Software reverse engineering · Software and its engineering → Compilers · Software and its engineering → Maintaining software

1 Introduction

The binary format (e.g., ELF but the same applies to MACHO or PE) is the defacto standard method of distributing software. It also is, however, essentially a black-box. Without source code available, it is hard to analyze a binary, to apply

This version of the contribution has been accepted for publication, after peer review (when applicable) but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. Use of this Accepted Manuscript version is subject to the publisher's Accepted Manuscript terms of use https://www.springernature.com/gp/open-research/policies/accepted-manuscript-terms.

patches to it, to test or to fuzz it. This is the cause of various security issues, undetected or unpatched vulnerabilities, and exploits. If we would distribute software in a binary format that is analyzable, editable, instrumentable, and fuzzable, that could dramatically impact the safety of our software. We thus argue software should not be distributed as black-boxes. However, distributing source code is not always an option either: third-party closed source software is ubiquitous. Enterprises need the capability of distributing software without publishing their source code or revealing proprietary internals. The question then is: what would a *middle ground binary format* look like that on one hand is "easier" to analyze and test, and "more instrumentable", but that on the other hand does not require one to distribute the source, nor makes it substantially easier to retrieve the source through decompilation?

First, let us consider this question practically with some examples. In order to do an analysis (e.g., verify memory-related properties or the absence of overflows), one needs to have bounds on the sizes of memory regions, and one needs to be able to traverse the binary, which requires control flow graphs (CFG) of each function [1,2,3,4,5]. In order to apply a patch, or to instrument a binary for fuzzing, one needs to be able to insert instructions and data into a binary [6,7]. Currently, a stripped Executable and Linkable Format (ELF) file does not enable any of these actions. However, for these kinds of actions, one does *not* necessarily need variable names, type information, or anything related to the structure of the original source code.

To summarize these requirements, we argue the need for a middle ground binary format that is 1.) *traversable*, 2.) *memory-structured*, and 3.) *instrumentable*. Traversability means that given the current instruction, we have a reasonably tight overapproximation of what the next instruction to be executed is. This is a challenge, since in binaries the address of the next instruction may be computed dynamically at run-time (indirections). Memory-structured means that compound contiguous memory segments such as stack frames (where local data is stored) and data sections (where global data is stored) can be decomposed into bounded independent memory regions. Without this property, analysis tools must treat memory segments as single undifferentiated arrays of bytes, thwarting verification of memory-related properties. Instrumentable means that the binary can be edited by adding instructions and data.

Based on these requirements, we argue that a middle ground binary format at least should provide *symbolized assembly* [8,9]. Symbolization replaces absolute addresses and offsets with symbolic labels. If the text- and data sections of a binary are symbolized, this enables insertion of new instructions and data, required for patching and instrumentation. Symbolization, however, is not enough. Symbolized assembly still can be untraversable due to indirections, and need not be memory-structured: information on which parts of a memory segment are to be considered independent can be lost. We will define *fully symbolized assembly* as symbolized assembly with additional characteristics that ensure the above three properties.

Retrieving fully symbolized assembly from a normal binary requires the solving of various problems that are *undecidable*. First of all, it requires disassembly, which in itself is already undecidable. Subsequently, it requires at least indirection resolving, function boundary detection, call graph construction, per-function termination analysis, and CFG construction just to obtain a traversable intermediate representation (IR). Then, recovering a structured memory model for stack frames and data sections requires advanced forms of pointer and shape analysis, or analysis based on probabilistics or heuristics [1,10,11,12,13,14]. Simply establishing which calling convention was used during compilation of the binary already requires a data flow analysis. All of these problems are undecidable [15]. However, all of the answers were known at build time. Rather than attempting to recover this lost information through analysis, our approach sidesteps these undecidable problems entirely by simply preserving it.

To emphasize this point more, we state that solving the above problems is not just theoretically undecidable, but that in practice state-of-the-art tools also often fail [16,17,18,19]. This holds true even though a large body of both academic literature as well as industrial tooling exists, in the fields of binary analysis, reverse engineering, and decompilation. Even today's state-of-the-art tools will not reliably disassemble and lift all binaries, are not capable of resolving all indirections, and do not produce recompilable code in a trustworthy fashion. This statement holds even when we only consider benign binaries produced by compilers. The core of the issue is that these tools have the impossible job of recovering information that was known – but lost – while building the binary.

The contribution of this paper is a binary format we call the *Executable, Linkable, and Liftable Format (ELLF)*. An ELLF is an ELF, augmented with a minimum amount of metadata that prevents the need to solve any of the above problems, or in other words, that makes the above problems decidable. We first discuss what exact metadata needs to be added to achieve this. We show how this metadata can be gathered at build time and how it can be added to an ELF without significant overhead in terms of compilation time and binary file size. We are agnostic of the compiler settings: one can still compile the code as one likes (e.g., at any optimization level). We then show that the above problems are indeed decidable by providing executable and efficient algorithms. To the best of our knowledge, this is the first effort on generating ground truth information at build time, packing it into a binary format, evaluating its correctness, *and* demonstrating that it is sufficient for lifting to a recompilable higher-level IR.

Currently, there are three ways to distribute an ELF: stripped, unstripped, or augmented with DWARF debugging information. First, note that all of these formats contain *too little information* to make any of the above problems decidable. For example, even for unstripped ELF files with DWARF information, recovering control flow is undecidable. Secondly, note that unstripped ELFs – with or without DWARF debugging information – also contain *too much* information. They contain – among others – names of global variables and internal functions, which is why many vendors choose to distribute stripped binaries instead. DWARF in particular includes a large amount of information that is

Stripped Binary

```
0x4000:
  push   rbp
0x4001:
  mov    rax, 0x4007

0x4004:
  jmp    rax
0x4006:
  sub    al, bl


0x5000:
  .long  0x4003
```

ELF + DWARF

```
0x4000: ; <main>
  push   rbp
0x4001:
  mov    rax, 0x4007
;       = <main>+0x07
0x4004:
  jmp    rax
0x4006:
  sub    al, bl


0x5000:
  .long 0x4003
;       = <main>+0x03
```

ELLF

```
main:
  push   rbp

  mov    rax, done

  jmp    rax

  .byte  0x2A
done:
  ret
g_var:
  .long 0x4003
;     not an address!
```

Fig. 1: Disassembly comparison.

irrelevant in this context and can be considered undesirable in the final binary, such as mappings from binary locations to their original source code locations. There is no format that is as close as possible to a stripped ELF, but that *does* provide crucial and security-relevant information on, e.g., the intended jump targets of indirections, or on which addresses in the binary are intended to be executable instructions and which not.

Example. Figure 1 shows a comparison of disassembling a stripped ELF, disassembling an ELF with DWARF, and disassembling an ELLF. When only the raw bytes of the binary are available, it is impossible to distinguish between instructions and raw data bytes, possibly causing disassemblers to misinterpret them. With DWARF debugging information, some of these problems can be alleviated as the entries of functions are often known and values can be compared to known pointer addresses. However, this approach is still prone to misinterpretation as soon as text and data are mixed. Additionally, if raw values coincide with known addresses, these raw values can be misinterpreted to be pointers. In the ELLF, such misinterpretations do not occur, as it contains the exact instruction locations and marks values as either raw values or pointers.

To summarize, the contributions of this work are:

- A study of which metadata is needed in order to make a binary traversable, memory-structured, and instrumentable.
- A tool to collect said metadata at build time, pack it into an efficient representation, and insert it as a section into the binary, producing ELLF files.[3]
- A lifter, implemented as an extension to FoxDec[4], that consumes an ELLF file and lifts it to correct fully symbolized assembly code.

[3] `https://github.com/ssrg-vt/ELLF`
[4] `https://github.com/ssrg-vt/FoxDec`

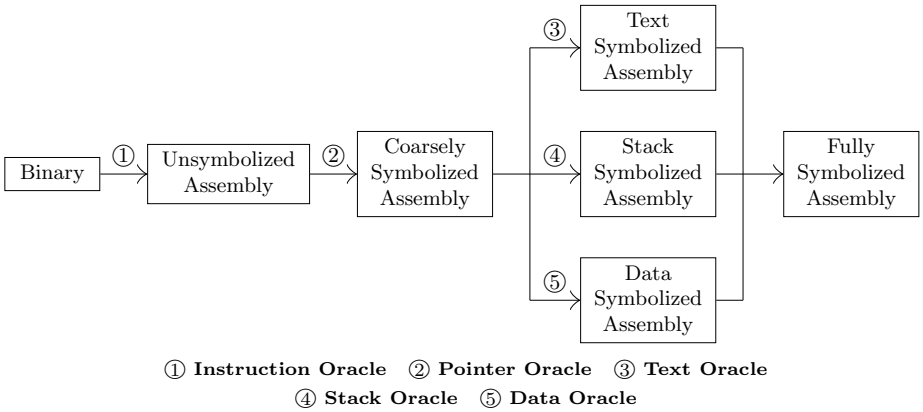

Fig. 2: Overview of the lifting pipeline.

Section 2 introduces the structure of the metadata we embed into binaries and explains how it enables precise disassembling and symbolization. Section 3 describes the metadata generation in more detail, including how the relevant information is extracted from the compiler outputs, how it is encoded, and how it is later consumed to disassemble stripped binaries. Section 4 presents our evaluation, in which we compile the LLVM Test Suite and embed ELLF metadata into the binaries. After stripping the binaries, we disassemble them using only our metadata and the binary itself. We then reassemble the disassembled output using a standard `gcc` and verify that the recompiled binaries match the behavior of the original binaries across the available test suites. This validates not just the completeness of the metadata but also the correctness of the disassembly. Section 5 discusses related work in extracting disassembling ground truth from the compiler and in lifting binaries to a rewritable format. This paper concludes with a discussion in Section 6.

2 ELLF Metadata

Our metadata design is informed by exploring the needs of a lifter capable of taking a binary and producing fully symbolized assembly. The lifter works in five steps, each raising the level of abstraction (see Figure 2). Each step is in itself undecidable, and therefore requires an *oracle*: information that makes the step decidable. Per step, we provide details on what the input and output is, what oracle is needed and why the oracle is necessary to make the step decidable. In this section, we provide an *abstract view* of the information provided by the oracles, or in other words, a specification. The concrete implementation will differ, mostly because storing the information naively will take up a lot of space. Section 3 provides more information on the actual concrete metadata inserted into the binary.

The first two steps have to be applied sequentially. The steps on text-, data- and stack symbolization can be executed independently in any order. Once all three have been achieved, this produces fully symbolized assembly.

We use $\mathbb{W}_{64}$ to denote the set of 64-bit words. Notation $X?$ is used to denote the set $X \cup \bot$ with $\bot$ a fresh element representing "None". Notation $f :: X \rightharpoonup Y$ denotes that f is a partial function from set X to set Y.

Step I: unsymbolized assembly. Step I takes as input raw data and outputs disassembled instructions (see Figure 4). It is one of the primary challenges in disassembling to determine the exact locations of instructions, and this is well-known to be undecidable [20,21,22,23]. Although the function boundary information available in DWARF can provide some guidance, identifying all instruction addresses remains undecidable.

The instruction oracle, abstractly seen, provides a set of addresses that are intended by the compiler to be the starting addresses of executable instructions. We model the instruction oracle as follows:

$$\textbf{InstructionOracle} :: \{\mathbb{W}_{64}\}$$

In other words, the oracle is a set of 64-bit values.

With the instruction oracle in place, lifting to unsymbolized disassembly becomes straightforward: a disassembler can traverse all addresses provided by the oracle, disassemble the instruction at the given address or emit the raw data if no instruction is at that address.

Step II: coarsely symbolized assembly. Step II takes as input unsymbolized assembly and lifts it to a representation where all pointers to text- and data-sections are replaced with labels (see Figure 4). The pointer oracle is modeled as:

$$\textbf{PointerOracle} :: \mathbb{W}_{64} \rightharpoonup \mathsf{Instr}\ [\mathbb{W}_{64}?] \mid \mathsf{Data}\ \mathbb{W}_{64} \mid \mathbb{W}_{64} - \mathbb{W}_{64}$$

The pointer oracle is a partial mapping from 64-bit addresses to pointer information. If the given address a belongs to an instruction, it provides an ordered list that gives for each operand either $\bot$ (if the operand is not a pointer), or a pointer value. In Figure 4, the `lea` instruction has two operands, where the second one is a pointer. It may also be the case that the given address a falls in some data section, and that at the location some pointer value p is stored. In that case **PointerOracle**(a) will return $\mathsf{Data}\ p$. Finally, it may be the case that address a falls in some data section and at that location the difference between two pointers is stored. This typically happens when compilers implement jump tables.

Retrieving this oracle from a binary is again undecidable. If the binary is not RIP-relative, it is undecidable to determine whether, e.g., the value occurring in the instruction `mov rax, 0x4000` is a pointer or not. In case the binary is RIP-relative, this problem does not occur. However, even then the cases where the origin of a value is the difference between two pointers are undecidable.

With this oracle in place, symbolization of pointers becomes straightforward. A disassembler can traverse all instruction operands and replace them by a

symbolic name if they are pointers, and traverse the data sections in similar fashion as well.

This lifting step produces coarsely symbolized assembly: one can move entire sections to other addresses, but within a section instructions and data are still fixed to their locations. As a result, one can already insert new text- or data sections. To actually intersperse new instructions within an existing text section, requires more fine-grained symbolization.

Step III: text symbolized assembly. Step III takes as input coarsely symbolized assembly and produces assembly where the text sections have been symbolized in a fine-grained fashion (see Figure 4). This step essentially introduces *basic blocks* into the code: blocks of instructions that are always executed sequentially. The oracle is modeled as:

$$\textbf{TextOracle} :: \mathbb{W}_{64} \rightharpoonup \mathsf{FunctionStart} \mid \mathsf{FunctionEnd} \mid \mathsf{BB}$$

If **TextOracle**(a) is FunctionStart, then address a is a function entry, i.e., the address of the first instruction of some function. FunctionEnd means at address a is the last instruction of a basic block that ends the function. This instruction thus can be a return statement, can halt the program, or can call some function that exits the program or does not return. BB means that the address is the starting address of some basic block that is not the start or end of a function. Again, deriving this oracle from a binary is undecidable, in this case because it requires at least resolving of indirections and termination analysis.

Having fine-grained symbolized text sections ensures that 1.) the function boundary (i.e., entry and exit points) of each function in the binary is known, and 2.) the CFG of each function is known. Both are crucial in enabling instrumentability and traversability of the binary. At this stage, one can intersperse new instructions (e.g., emit a logging message or insert a null-pointer check).

Step IV: stack symbolized assembly. The stack frames of functions can be symbolized in fine-grained fashion as well. A stack frame is a contiguous memory segment used for storing local objects such as variables, arrays and structs. Up to this step, there is no information available on how the stack frame is structured, i.e., how the contiguous memory can be broken down into objects. It is a notoriously hard and undecidable problem to recover structure of stack frames from a binary [14].

The stack oracle is modeled as:

$$\textbf{StackOracle} :: \mathbb{W}_{64} \rightarrow \{\mathbb{W}_{64}\}$$

When some function `f` is called with entry address a, the stack pointer will have some initial value rsp_0. The stack frame then stores local data at various offsets (see Figure 3). For this example, the oracle will return the set $\{4, 32, 40\}$.

Having fine-grained symbolized stack frames provides analysis tools with information necessary to do their work. By inserting the stack frame structure into the binary, one can, e.g., shuffle the objects on the stack. This is a form of binary diversification [24,25], a practical technique to prevent bugs from being exploitable. One can also take a binary with shuffled stack frames and fuzz it,

Fig. 3: Example of stack frame structure.

to see if any behavior changes with respect to the original. If so, that would indicate some out-of-bounds access, or at the least some violation of how the stackframe was intended to be structured by the compiler. Such violations are well-known to be the source of vulnerabilities. Finally, static analysis tools can do bounds checking, but this requires them to *know* the bounds of local arrays. That is exactly the information provided by this oracle.

Step V: data symbolized assembly. In similar fashion to achieving fine-grained stack symbolization, the data sections can be symbolized in fine-grained fashion as well (see Figure 4). The data oracle provides this information, and is modeled as follows:

$$\textbf{DataOracle} :: \mathbb{W}_{64} \rightharpoonup \mathbb{N}^{+}$$

It maps addresses to positive integers. If, e.g., **DataOracle**(a) returns *80*, then the 80-byte region at address a is intended by the compiler to be an independent object. Thus, after fine-grained data symbolization, one should be able to move and shuffle global objects around in memory without changing the intended behavior of the binary, in similar fashion to stack symbolization.

3 ELLF Generation

All the information that is needed to generate the oracles is available at some time during building, some of the information is preserved in the resulting ELF/DWARF file. We require the compiler to output some extra information that is not generated by default. We modify the compiler arguments to include `-g` to always generate DWARF information, `-fbasic-block-address-map` to emit information about the basic blocks in the binary, `--emit-jump-table-sizes-section` to detect which pointers belong to the same jump table, and the linker arguments to include `-Wl,-Map=<name>.map` for a mapping from the original object files to their locations inside the binary. The debugging information is discarded after metadata generation as it is not needed in the final binary or for disassembling. Additionally, we add symbols to the start of every section and store relocations to these symbols to improve the reliability of the linker map.

Table 1 contains the sources for the metadata for each step from Section 2. Each source is annotated by the stage in which it is available: Can we get the information in a standard ELF file (*ELF*), only binaries with debugging information (*DWARF*) or is the information only available during compiling and linking (*CC/LD*)?

Disassembling. For basic disassembling, we need to know the addresses of all instructions in the binary. This is information that is not available by default

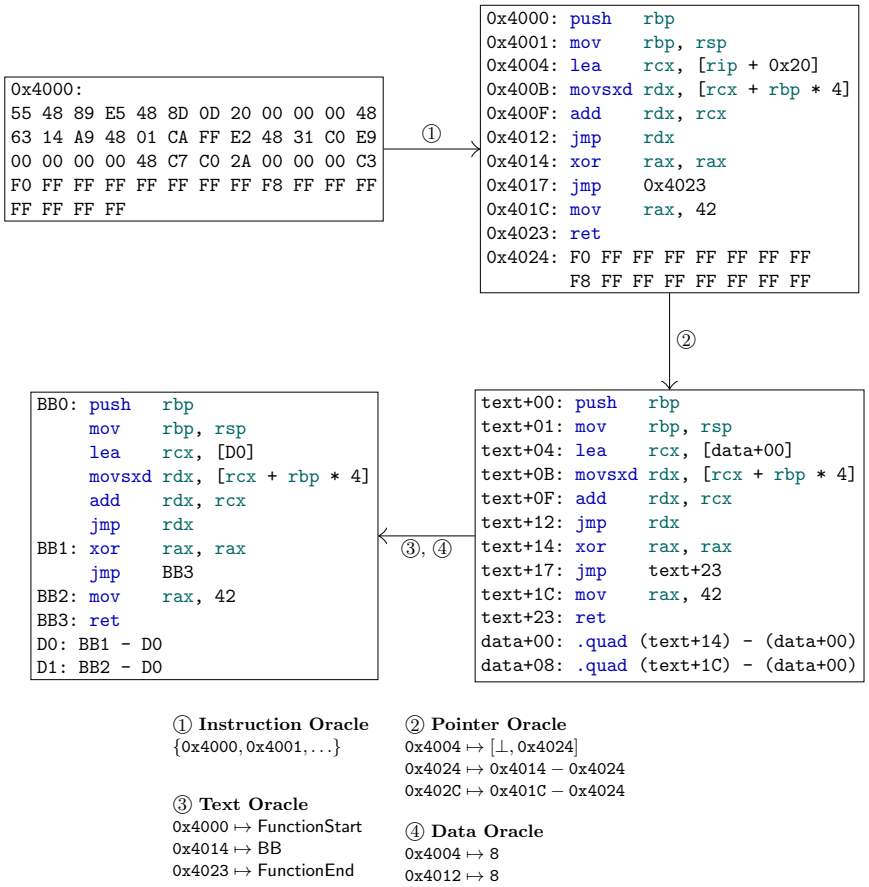

Fig. 4: Example of lifting, with examples of the data provided by the oracles.

during any stage of compilation. Clang supports an option to emit information about the basic blocks in the binary. Inside a basic block, all bytes must be instructions since there are no jumps to skip over some bytes. Conversely, every byte that does not fall within any basic block must be padding or data. From this information, we generate the **InstructionOracle**. Storing the address of every instruction in the metadata would be very expensive. Instead, we compress this information into instruction regions that contain the address of the first instruction and the number of uninterrupted, sequential instructions. This information is more coarse than the basic block information, multiple basic blocks can be collapsed into a single instruction region. Additional metadata that is necessary to recompile the binary, such as the dynamic libraries required, is available in the standard ELF headers and does not need to be included in the metadata.

Coarse Symbolization. For the coarse symbolization, we need to know which operands in instructions and which bytes in the `.data` sections are point-

Table 1: Sources of disassembling and symbolization information in ELF/DWARF binaries. The columns ELF, DWARF, CC/LD indicate if the information is collected via the final ELF binary, the debugging information in the final ELF binary or during the compilation and linking phase. Symbols: ✓ = available, ✓✓ = available even after `strip`ing, △ = partially available.

Purpose	**Information**	**Present In**			**Location**
		ELF	**DWARF**	**CC/LD**	
Disassembling	Instruction Addresses			✓	`.llvm_bb_addr_map`
	`_start` Symbol Address	✓✓			ELF Header
	Required Dynamic Libraries	✓✓			`.dynamic`, `.dynstr`
Coarse Symbolization	Pointer-To-Text Operands	✓✓[1]		✓	`.text`, `.rel.*`, `.rela.*`
	Pointer-To-Data Operands	✓✓[1]		✓	`.text`, `.rel.*`, `.rela.*`
	Pointer-To-Text Variables		△	✓	`.debug_info`, `.rel.*`, `.rela.*`
	Pointer-To-Data Variables		△	✓	`.debug_info`, `.rel.*`, `.rela.*`
Text Symbolization	Internal Function Addresses	✓			`.symtab`
	Statically Linked Function Addresses[2]	✓			`.symtab`
	Dynamically Linked Function Addresses	✓✓			`.rel.*`, `.rela.*`, `.plt`, `.dynsym`, `.dynstr`
	Internal Label Addresses	✓✓[1]		✓	`.text`, `.llvm_bb_addr_map`
	Jump Table Information	△		✓	`.rel.*`, `.rela.*`, `.llvm_jump_table_sizes`
Stack Symbolization	Local Variable Locations		✓		`.debug_info`
	Local Variable Types[3]		✓		`.debug_info`
Data Symbolization	Global Variable Addresses	△	△	✓	`.debug_info`, `.rel.*`, `.rela.*`
	Global Constant Addresses	△	△	✓	`.debug_info`, `.rel.*`, `.rela.*`, `.symtab`
	String Addresses		△	✓	`.debug_info`, `.symtab`
	Global Variable Types[3]		✓		`.debug_info`
Exception Support	CFI Directives	✓			`.eh_frame`
	Language Specific Data Areas	△		△	`.eh_frame`, `.llvm_bb_addr_map`

[1]: In position dependent binaries, this information is not known.
[2]: Due to inlining, this information does not always apply.
[3]: Only the sizes of the types are needed.

ers to other locations, and which are pure data. In a position independent binary, resolving which instructions contain pointers to symbols is trivial as they do not use absolute addresses. In position dependent binaries, this information is accessible through *relocations*. After compilation, the final addresses of functions and other objects are not known. Instead, the instructions and variables accessing this information are filled with zero bytes to be filled in during linking. Relocations are added as hints for the linker to know which zero bytes need to be replaced by the addresses of objects. After linking, the relocations are often no longer needed and discarded.

Clang supports the `--emit-relocs` flag to keep relocations from being discarded. However, there have been bugs in `lld` where relocations got lost even if this flag is passed. For this reason, we collect this information directly from the object files, where the relocations are necessarily present and map their addresses into the binary using the linker map file.

If a datum is a pointer according to this metadata then its original type in C/C++ must have been a pointer as well, or it was the target of a function call or control flow jump. But not every C/C++ pointer is also in the pointer oracle. The value of a variable like `volatile int *dst = 0x6000` is a pointer but it must not be symbolized during the coarse symbolization step.

Text Symbolization. For the text symbolization, we need to add a label to all instructions that are intended as jump targets. The only instructions that the compiler intends as jump targets are the first instructions of basic blocks. This information is read directly from LLVM's basic block address map. Additionally, to generate correct assembly code during recompilation, we need to know which instruction start and end functions. The starts of functions are known via the symbol table in standard ELF but are more reliably known using the basic block address map. For the text metadata, we only store the addresses where basic blocks start and a marker for the basic block that starts a function. All other information is deduced from there.

Data Symbolization. For the data symbolization, we need to add a label to all global variables. This information is not available in ELF, but debugging symbols in DWARF contain debugging information entries for global variables and their types. Unlike the text symbolization, it is not enough to know the starting address of global variables to generate valid assembly code for them. Binaries reuse strings in order to reduce their memory footprint, for example, if a program uses the strings "`Hello World`" and "`World`", the binary will only store the former and access the latter via an offset. Still, DWARF sees these as conceptually distinct variables and contains incorrect information. To combat this, we do not keep variables that would overlap with previous ones. If DWARF misses a variable, our data oracle becomes slightly less precise but remains sound. The data that should belong to a missed variable is merged with the variable directly preceding it.

Stack Symbolization. For the stack symbolization, we need to add a label to all local variables. This is done in similar fashion to the data symbolization. The main difference is that local variables belong to a function and are not

available everywhere in the binary. As such, the metadata for stack symbolization cannot contain purely the addresses of local variables, it needs to pair them with the address of the function containing the local variables.

Exception Support. C++ binaries may define, throw and catch exceptions. On the binary level, this is implemented via DWARF Call Frame Information (CFI) entries in the `.eh_frame` and the Language Specific Data Area (LSDA) in the `.gcc_except_table`. On the assembly level, the CFI entries are represented as directives and interspersed with normal instructions. These can be directly reconstructed from the `.eh_frame`. The LSDA needs to be symbolized and references labels from the text metadata for its landing pads and labels from the data metadata for the virtual tables of the exception types. Overall, exceptions do not need any additional metadata that is not already covered by previous ones.

4 Evaluation

We evaluate our approach with respect to *correctness* and *overhead* to validate that the metadata described in Section 3 enables trustworthy disassembly in practical settings. Our evaluation uses 200 programs from the LLVM test suite, a widely used benchmark for assessing functional correctness, compilation performance, and binary size. Correctness is evaluated by executing the test cases. Overhead is quantified in terms of compilation time, resulting binary size, and execution time of the generated binaries. The benchmark set comprises C and C++ programs and reflects realistic programming patterns, including exception handling and compiler optimizations that are traditionally challenging for disassembly, such as jump tables.

Our evaluation considers three binary variants: (1) baseline binaries compiled with `clang-21`, (2) metadata-augmented binaries produced by `ELLF clang`, and (3) binaries obtained by disassembling and recompiling the metadata-augmented binaries.

4.1 Experiment

A Docker-based environment to reproduce the evaluation results is available online.[5]

We use the *MultiSource* benchmarks from version `22.1.0-rc2` of the LLVM test suite.[6] Our evaluation includes all 198 programs that successfully compiled under the baseline `clang` configuration and our compilation flags. Two programs were excluded because they either failed to compile with the baseline compiler or were incompatible with the selected compilation settings.

While the LLVM test suite provides a standardized and widely used benchmark for compiler evaluation, it primarily consists of relatively small, self-contained

[5] `https://osf.io/4ebds/`

[6] `https://github.com/llvm/llvm-test-suite/archive/llvmorg-22.1.0-rc2.tar.gz`

Table 2: Comparison between standard `clang` and our `ELLF clang`. Results are obtained by compiling the LLVM test suite in `Release (-O2)` mode for both `clang` and `ELLF clang`, and `RelWithDebInfo (-O2 -g)` mode for `clang`. `Release` mode is taken as the base for the metrics. Shown are the compile-time metrics "Compile Time" and "Binary Size", and the run-time metrics "Exec Time" and "Test Result" as obtained by the LLVM `lit` tool.

Name	**Compile Time** (s)		**Binary Size** (KiB)			**Exec Time** (ms)		**Test Result**	
	`clang` `-O2`	`ELLF` `-O2`	`-O2`	`clang` `-O2 -g`	`ELLF` `-O2`	`clang` `-O2`	`ELLF` `-O2`	`ELLF` original	`ELLF` recompiled
alacconvert-decode	17.2	26.7 (155.0%)	64	178 (277.4%)	75 (115.8%)	18	19 (105.0%)	PASS	PASS
alacconvert-encode	16.8	27.2 (161.9%)	64	178 (277.4%)	75 (115.8%)	25	24 (99.2%)	PASS	PASS
burg	25.8	43.6 (168.8%)	89	218 (244.8%)	113 (126.5%)	13	13 (101.6%)	PASS	PASS
clamscan	141.7	222.6 (157.1%)	977	2297 (235.0%)	1104 (113.0%)	82	81 (98.9%)	PASS	PASS
SIBsim4	7.3	10.3 (141.3%)	69	149 (217.5%)	74 (107.6%)	1001	1000 (99.9%)	PASS	PASS
SPASS	83.7	125.3 (149.7%)	787	2911 (369.8%)	838 (106.5%)	3320	3377 (101.7%)	PASS	PASS
aha	2.2	3.8 (173.0%)	15	24 (162.1%)	17 (116.0%)	696	697 (100.1%)	PASS	PASS
make_dparser	19.0	29.0 (152.6%)	404	665 (164.6%)	491 (121.5%)	22	24 (110.9%)	PASS	PASS
hbd	26.9	46.0 (170.7%)	68	248 (365.8%)	97 (143.9%)	14	13 (94.4%)	PASS	PASS
hexxagon	9.0	13.1 (145.3%)	36	105 (287.0%)	44 (120.6%)	3965	3820 (96.3%)	PASS	PASS
...	...	...	...	...	...	...	...	...	...
rawcaudio	2.2	3.6 (166.7%)	8	15 (174.9%)	10 (119.8%)	10	10 (98.0%)	PASS	PASS
rawdaudio	2.1	3.8 (179.6%)	8	15 (174.8%)	10 (119.8%)	7	10 (134.7%)	PASS	PASS
encode	6.8	11.6 (169.2%)	19	41 (218.6%)	23 (122.3%)	27	28 (104.9%)	PASS	PASS
toast	24.9	42.9 (172.5%)	74	187 (252.1%)	88 (118.3%)	18	18 (100.0%)	PASS	PASS
cjpeg	72.2	117.5 (162.7%)	225	760 (338.4%)	254 (113.1%)	12	14 (119.3%)	PASS	PASS
mpeg2decode	20.4	33.2 (162.9%)	92	182 (199.4%)	108 (118.3%)	16	14 (91.6%)	PASS	PASS
nbench	7.8	12.3 (156.8%)	75	165 (221.4%)	84 (113.2%)	621	641 (103.2%)	PASS	PASS
sim	2.1	2.9 (142.7%)	30	55 (186.0%)	32 (108.5%)	1257	1257 (100.0%)	PASS	PASS
frame_layout	2.7	4.4 (164.5%)	145	937 (644.6%)	193 (132.9%)	16	17 (106.1%)	PASS	FAIL
rounding	1.3	2.9 (217.4%)	13	19 (146.2%)	15 (114.6%)	8	9 (112.3%)	PASS	PASS
Total	2875.1	4504.6 (156.7%)	22534	58126 (258.0%)	28523 (126.6%)	181123	178261 (98.4%)	198	197

programs. To assess the behavior of our approach in more realistic settings, we additionally performed automated testing on substantially larger real-world applications, including OpenALPR (a license plate recognition system), NASA's Core Flight System, custom image processing servers built on the STB image library, and ROS2 (Robot Operating System 2). For these programs, we compared the behavior of original and reconstructed binaries across a range of valid and invalid inputs and did not observe any deviations. Moreover, the overheads in compilation time and binary size were consistent with those observed for the LLVM test suite, suggesting that our results generalize beyond the benchmark set.

We establish a baseline by compiling all programs in `Release` mode using `clang-21`. In addition, we compile the programs in `RelWithDebInfo` mode to quantify the relative overhead introduced by DWARF debugging information.

To measure the overhead of our approach, we compile the same programs in `Release` mode using our `ELLF clang` wrapper, which invokes the same `clang-21` while generating metadata for the resulting artifacts. This allows us to measure the overhead introduced by metadata generation in terms of compilation time and binary size.

To ensure that metadata generation does not alter the binaries' behavior, we execute the full LLVM test suite using the `lit` tool[7] on both the baseline and metadata-augmented binaries. This verifies that the addition of metadata does not alter functional behavior. The reported execution times additionally allow us to quantify the runtime impact of metadata integration.

To evaluate the disassembly stage, we process the metadata-augmented binaries using a fork of the FoxDec [26] disassembler which we modified to consume our metadata. The resulting assembly files are recompiled using `gcc-13`. We then re-run the `lit` test suite on the recompiled binaries to verify that (1) the disassembly produces valid assembly code and (2) the reconstructed binaries preserve the original program behavior.

The metrics collected by `lit` are summarized in Table 2.

Functional Correctness. All 198 original programs pass all test cases. Similarly, all programs compiled with our toolchain also pass all test cases, demonstrating that the addition of metadata does not alter functional behavior. The only exception in the disassembly-and-recompile stage is the program `frame_layout`, which fails a single test case. This failure is attributable to a rare linker interaction not yet supported by our disassembler rather than insufficient metadata. These and other limitations are discussed in detail in Section 4.2. Overall, these results indicate that our metadata is sufficient to produce recompilable assembly code and that the resulting binaries faithfully preserve the behavior of the original programs.

Compilation Time. Compiling with our toolchain introduces an average overhead of 57% relative to baseline compilation. Detailed profiling shows that approximately 7% of compilation time is spent on metadata generation and 5%

[7] `https://github.com/llvm/llvm-project.git#subdirectory=llvm/utils/lit`

on pre- and post-processing of object files; the remainder is spent on standard `clang` execution.

Binary Size. Adding ELLF metadata increases binary size by 27%. By comparison, including DWARF debug information incurs a larger overhead, increasing binary size by 158%.

Execution Time. Execution times of metadata-augmented binaries are essentially identical to the original binaries. Minor observed variations fall within expected measurement noise, with deviations of up to ±10% relative to the baseline. Notably, we observe a small average speedup of 1.6% for metadata-augmented binaries. This result is based on averaging five independent runs of the LLVM test suite. Since our approach does not modify the generated machine code, we attribute this difference to measurement noise and normal system variability rather than a systematic performance improvement.

4.2 Limitations

Original Assembly Code

```
.P1: .quad .L1
.P2: .quad .L2
.L1: .asciiz "Hello World"
.L2: .asciiz "World"
```

After Merging

```
.P1: .quad .L1
.P2: .quad .L1+6
.L1: .asciiz "Hello World"
; .L2 got suffix merged
```

Original Assembly Code

```
.P3: .L3
.P4: .L4
.L3: .quad some_var
; Assuming the address of some_var
; starts with 0xC0FFEE
.L4: .quad 0xDEADBEEF00C0FFEE
```

After Merging

```
.P3: .L3
.P4: .L3+4
.L3: .quad some_var
     .long 0xDEADBEEF

; .L4 got suffix merged
```

Fig. 5: Example of suffix merging. On the top: Suffix merging of string data. On the bottom: Suffix merging of pointer data with raw bytes.

Our toolchain does not yet fully support some rare binary patterns. In particular, the disassembler cannot handle merged sections of different types, and certain unusual switch table layouts are not fully captured by our metadata generation.

Merged Sections. Linkers may merge multiple read-only data sections when they contain identical content or when one is a suffix of another, similar to string merging based on suffixes. Consider the top example in Figure 5: in the original assembly, the strings “Hello World” and “World” exist as separate objects (`.L1` and `.L2`). After merging, `.L2` is removed and referenced as `.L1+6`. This does not affect our disassembler, which does not need to distinguish these objects.

The bottom example in Figure 5 illustrates a more challenging case. If a symbolized pointer (the value of `.L3`) overlaps with the beginning of the next object (`.L4`), the linker may merge the two sections. In this scenario, the value of the merged object depends on the linker's placement choices, which can differ in the recompiled binary from the original binary. Such overlaps caused the failure of the `frame_layout` test, and resolving them would require a more sophisticated implementation in the disassembler.

Switch Tables. High-level switch statements can be lowered by the compiler in multiple ways. Small or sparse switches may be implemented as chains of `cmp` instructions, while larger switches are often lowered to jump tables. Our toolchain fully supports standard jump tables, extracting the necessary metadata from the `.llvm_jump_table_sizes` section.

Switch statements that involve many strings are lowered to lookup tables with pointers, which are also supported by our metadata generation. However, in rare cases, compilers generate lookup tables with larger objects for which relocations do not follow a predictable pattern. In such cases, the metadata does not provide all information required for correct code generation. In particular, the disassembler needs to generate pointer values of the form `.Lnth_entry - .Lfirst_entry`. This requires knowing which entries belong to the same table, this information is not always known. While this did not cause any failures in the LLVM test suite, it could potentially affect other programs with unusual switch layouts.

5 Related Work

Traditionally, disassembling is done by using heuristics to work around the inherently incomplete information in a binary [20,16,18]. While these approaches are able to achieve impressive results, they can never be 100% correct. We will examine other work that focuses on capturing metadata at build time, and work that aims to rewrite binaries correctly.

Capturing metadata at build time. Alves-Foss et al. [27] note that the existing information in ELF/DWARF is insufficient to be used as a ground truth for binary disassembling. They propose recommendations for what a correct ground truth for disassembling should entail such as the instructions in the `.text` sections or the boundaries of functions. Our metadata is largely compatible with these recommendations. We found that for disassembling, not all their requirements are necessary, such as markers for `noreturn` functions but agree that these are useful for further analyses. They say that tracking relocations via compiler modifications is not a feasible long term solution. We found that tracking relocations to resolve pointers from the object files works well in practice, even for position independent code.

Pang et al. [28,18] modify the `clang` compiler to emit information such as the instructions written by the code generation and collect the relocations to distinguish between symbolized pointers and raw values. Tracking the compilation from inside the compiler has the advantage that the result is guaranteed to

be correct but it also makes this approach harder to maintain. Any changes to `clang` can break this instrumentation. Currently, only an older version of LLVM is supported, making it impossible to generate ground truth for C++20 binaries.

Li et al. [29] leverage the listing files emitted by `gcc` and other compilers to generate their disassembling ground truth. Compared to relying on compiler modifications, this makes their approach applicable to all compilers that are able to emit listing files and is more resilient towards changes by the compiler maintainers. As they remark themselves, this also excludes them from using certain compiler features, such as link time optimizations.

Ince et al. [30] instrument the basic block structure used during compilation and embed the control flow structure into the binary. They use this information to aid tools such as `Dyninst` [31] as they no longer need to recover the CFG using static analyses. Apart from this information (which is comparable to our text metadata), they do not embed any other metadata.

Binary Rewriting through Lifting and Recompiling. Several approaches aim to provide frameworks that lift a binary to a rewritable IR. Ramblr lifts binaries into reassemble assembly suitable for patching and instrumentation [8]. Their approach leverages techniques such as control flow graph recovery and content classification (differentiating code from data). King et al. propose a layout-agnostic binary rewriting tool called Egalito [32]. This tool uses standard disassembling heuristics to parse the binary into a rewritable format. As such, these tools are not able to produce correct recompiled binaries 100% of the time.

BOLT by Panchenko et al. is a post-link optimizer leveraging the LLVM framework [33]. It extracts a large part of its required information for disassembling directly from ELF information such as the symbol table and DWARF debugging information. They mention that relocations can optionally be tracked in the binaries by passing the `--emit-relocs` flag to the linker. The main differences to our tool are that we aim to minimize trust in information that could be wrong such as the addresses in the symbol table or the relocations tracked into the binary by extracting that information directly from the object files where it must be correct. Additionally, BOLT works on unstripped binaries with DWARF information and does the lifting internally while we add a minimal amount of information into stripped binaries and produce general purpose assembly.

6 Discussion And Conclusion

Can ELLF be used to distribute closed-source software? As explained in the introduction, we propose here a middle ground binary format that should make binaries less black-box, but that should also not make it significantly easier to decompile to source code than a normal stripped binary. We informally argue that this is indeed the case. Of course, for an ELLF several key problems that normally are required to be solved when doing decompilation are already solved. It is thus significantly easier to lift an ELLF to an IR that entails, e.g., which instructions are to be executed and which not, and which memory regions are

separate objects. That IR, however, is not close in level of abstraction to source code. Note that in the context of decompilation, not all source code is created equal. It is, e.g., possible to decompile to "untyped" C code (using type punning) with non-descriptive names and with all control flow implemented by `goto` statements. It is harder to decompile to well-structured, humanly understandable and largely architecture independent C code. With the term "decompilation" we here mean decompilation to C source code that tries to resemble code as it was originally written. Such decompilation requires several further steps that are *not* made easier:

Recovering humanly readable code. As an example, consider the following x86-64 assembly: `mov edx,0x1999999A; imul eax`. The humanly readable equivalent of this snippet is `edx := eax / 10` (32-bit integer division by ten). A decompiler can try to recover humanly readable constructs from the low-level implementations generated by a compiler. This work is not made easier by any of the metadata supplied by ELLF.

Recovering syntactic control flow. ELLF provides control flow implemented through jumps. To lift this to a representation with syntactic control flow structures such as if-statements and loops is non-trivial [34,35,36] (at least: not in a way that resembles humanly written code, otherwise it is always possible [37]). Even though ELLF provides the intended control flow, how to eliminate all `goto` statements is not made easier than it is for regular stripped ELF files.

Recovering types. This is a well-studied but hard problem [38,39,40,41]. Major existing binary-level tools such as IDA-PRO and Ghidra try to do type recovery, but in general they may produce unsound results, or omit types producing holes in the generated code. Type recovery is not made easier by the metadata added to the ELLF, other than that it is known how memory segments are structured into regions.

Is ELLF trustworthy even if DWARF is not? Our evaluation demonstrates correctness by showing that the lifted-then-recompiled binary behaves equivalent to the original. In contrast, correctness of the DWARF debugging format has been put into question [42,43,44]. At first glance, this may seem to contradict ELLF correctness, since some of the DWARF information is used to generate the ELLF metadata. However, when DWARF is incorrect it is typically *incomplete*, which has no implication on correctness of the ELLF metadata. We only rely on DWARF for obtaining information on variables. If DWARF is incomplete and misses a variable, fine-grained stack- and data symbolization becomes a bit coarser but not incorrect.

What if the ELLF metadata is incorrect or tampered with? Our approach assumes that the LLVM-based metadata faithfully reflects the compiler's internal knowledge, and that our tooling correctly interprets the resulting compiler artifacts. If the metadata were incorrect, whether due to a compiler bug or deliberate tampering, correctness of the lifting process could no longer be guaranteed. In our evaluation, metadata correctness is empirically supported by the fact that lifted-then-recompiled binaries pass 197 out of 198 test cases. Validation of the

metadata itself, as a safeguard against tampering, is currently not implemented and is an interesting direction for future work.

Does ELLF work for hand-written assembly? In general, our approach does not extend to hand-written assembly, since the metadata ELLF relies on is produced by the compiler pipeline and is unavailable for manually written code. Supporting arbitrary hand-written assembly is not the goal of this work. That said, some common uses of hand-written assembly integrate well. Embedding raw data via assembly, as commonly done in systems such as web servers, does not pose issues. When hand-written assembly code is present, it does not invalidate the entire binary; only those regions lack sufficient metadata and may fall back to heuristic disassembly, without correctness guarantees on that part.

Does ELLF only apply to ELF? This paper focuses on Linux ELF, but all concepts apply to the Windows PE format and MacOS Mach-O format as well. Specifically, Section 2 essentially describes a specification of what metadata should be generated at build-time to make the binary liftable to a handleable IR. Our effort is also largely compiler-agnostic: even though our implementation is LLVM based, a similar approach could be implemented for various compilers.

Conclusion. Within the Linux community, there is growing interest in updating the ELF format. It is one of the topics within The Open Source Security Foundation (OpenSSF) community of the Linux Foundation, which has started the Reliable Software Decomposition Special Interest Group (SIG) on this topic. With this paper, we aim to provide such an effort with a fundament.

Acknowledgements This paper is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) and Naval Information Warfare Center Pacific (NIWC Pacific) under Prime Contract No. N66001-21-C-4028, and by DARPA under Prime Contract No. HR001124C0492. Any views, opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and should not be interpreted as presenting the official policies or position, either expressed or implied, of DARPA or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints of publications for Government purposes notwithstanding any copyright notation hereon.

References

1. Balakrishnan, G., Reps, T.: Analyzing memory accesses in x86 executables. In: International conference on compiler construction. pp. 5–23. Springer (2004). 10.1007/978-3-540-24723-4_2
2. Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Saxena, P.: BitBlaze: A new approach to computer security via binary analysis. In: Information Systems Security: 4th International

Conference, ICISS 2008, Hyderabad, India, December 16-20, 2008. Proceedings 4. pp. 1–25. Springer (2008). 10.1007/978-3-540-89862-7_1
3. Wang, T., Wei, T., Lin, Z., Zou, W.: Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In: NDSS. pp. 1–14 (2009)
4. Brumley, D., Jager, I., Avgerinos, T., Schwartz, E.J.: BAP: A binary analysis platform. In: International Conference on Computer Aided Verification. pp. 463–469. Springer (2011). 10.1007/978-3-642-22110-1_37
5. Djoudi, A., Bardin, S.: Binsec: Binary code analysis with low-level regions. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 212–217. Springer (2015). 10.1007/978-3-662-46681-0_17
6. Ben Khadra, M.A., Stoffel, D., Kunz, W.: Efficient binary-level coverage analysis. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 1153–1164 (2020). 10.1145/3368089.3409694
7. Dinesh, S., Burow, N., Xu, D., Payer, M.: Retrowrite: Statically instrumenting COTS binaries for fuzzing and sanitization. In: 2020 IEEE Symposium on Security and Privacy (SP). pp. 1497–1511. IEEE (2020). 10.1109/SP40000.2020.00009
8. Wang, R., Shoshitaishvili, Y., Bianchi, A., Machiry, A., Grosen, J., Grosen, P., Kruegel, C., Vigna, G.: Ramblr: Making reassembly great again. In: NDSS (2017)
9. Verbeek, F., Naus, N., Ravindran, B.: Verifiably correct lifting of position-independent x86-64 binaries to symbolized assembly. In: Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security. pp. 2786–2798 (2024). 10.1145/3658644.3690244
10. Balakrishnan, G., Reps, T.: Recovery of variables and heap structure in x86 executables. Tech. rep., University of Wisconsin-Madison Department of Computer Sciences (2005)
11. Kinder, J., Veith, H.: Precise static analysis of untrusted driver binaries. In: Formal methods in computer aided design. pp. 43–50. IEEE (2010)
12. Navas, J.A., Schachte, P., Søndergaard, H., Stuckey, P.J.: Signedness-agnostic program analysis: Precise integer bounds for low-level code. In: Asian Symposium on Programming Languages and Systems. pp. 115–130. Springer (2012). 10.1007/978-3-642-35182-2_9
13. Zhang, Z., Ye, Y., You, W., Tao, G., Lee, W.c., Kwon, Y., Aafer, Y., Zhang, X.: Osprey: Recovery of variable and data structure via probabilistic analysis for stripped binary. In: 2021 IEEE Symposium on Security and Privacy (SP). pp. 813–832. IEEE (2021). 10.1109/SP40001.2021.00051
14. Rose, A., Bansal, S.: Modeling dynamic (de) allocations of local memory for translation validation. Proceedings of the ACM on Programming Languages **8**(OOPSLA1), 1463–1492 (2024). 10.1145/3649863
15. Rice, H.G.: Classes of recursively enumerable sets and their decision problems. Transactions of the American Mathematical society **74**(2), 358–366 (1953). 10.1090/S0002-9947-1953-0053041-6
16. Andriesse, D., Chen, X., Van Der Veen, V., Slowinska, A., Bos, H.: An In-Depth analysis of disassembly on Full-Scale x86/x64 binaries. In: 25th USENIX security symposium (USENIX security 16). pp. 583–600 (2016). 10.5555/3241094.3241140
17. Djoudi, A., Bardin, S., Goubault, É.: Recovering high-level conditions from binary programs. In: International Symposium on Formal Methods. pp. 235–253. Springer (2016). 10.1007/978-3-319-48989-6_15

18. Pang, C., Yu, R., Chen, Y., Koskinen, E., Portokalidis, G., Mao, B., Xu, J.: SoK: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask. In: 2021 IEEE Symposium on Security and Privacy (SP). pp. 833–851 (2021). 10.1109/SP40001.2021.00012
19. Liu, Z., Wang, S.: How far we have come: testing decompilation correctness of C decompilers. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. p. 475–487. ISSTA 2020, Association for Computing Machinery, New York, NY, USA (2020). 10.1145/3395363.3397370
20. Schwarz, B., Debray, S., Andrews, G.: Disassembly of executable code revisited. In: Ninth Working Conference on Reverse Engineering, 2002. Proceedings. pp. 45–54. IEEE (2002). 10.1109/WCRE.2002.1173063
21. Wartell, R., Zhou, Y., Hamlen, K.W., Kantarcioglu, M., Thuraisingham, B.: Differentiating code from data in x86 binaries. In: Joint European Conference on Machine Learning and Knowledge Discovery in Databases. pp. 522–536. Springer (2011). 10.1007/978-3-642-23808-6_34
22. Smithson, M., ElWazeer, K., Anand, K., Kotha, A., Barua, R.: Static binary rewriting without supplemental information: Overcoming the tradeoff between coverage and correctness. In: 2013 20th Working Conference on Reverse Engineering (WCRE). pp. 52–61. IEEE (2013). 10.1109/WCRE.2013.6671280
23. Wartell, R., Zhou, Y., Hamlen, K.W., Kantarcioglu, M.: Shingled graph disassembly: Finding the undecideable path. In: Pacific-Asia Conference on Knowledge Discovery and Data Mining. pp. 273–285. Springer (2014)
24. Bhatkar, S., DuVarney, D.C., Sekar, R.: Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In: 12th USENIX Security Symposium (USENIX Security 03) (2003)
25. Bhatkar, S., DuVarney, D.C., Sekar, R.: Efficient techniques for comprehensive protection from memory error exploits. In: 14th USENIX Security Symposium (USENIX Security 05). pp. 271–286 (2005)
26. Verbeek, F., Olivier, P., Ravindran, B.: Sound c code decompilation for a subset of x86-64 binaries. In: Software Engineering and Formal Methods: 18th International Conference, SEFM 2020, Amsterdam, The Netherlands, September 14–18, 2020, Proceedings. p. 247–264. Springer-Verlag, Berlin, Heidelberg (2020). 10.1007/978-3-030-58768-0_14
27. Alves-Foss, J., Venugopal, V.: The inconvenient truths of ground truth for binary analysis. CoRR **abs/2210.15079** (2022). 10.48550/ARXIV.2210.15079
28. Pang, C., Zhang, T., Yu, R., Mao, B., Xu, J.: Ground truth for binary disassembly is not easy. In: 31st USENIX Security Symposium (USENIX Security 22). pp. 2479–2495. USENIX Association, Boston, MA (Aug 2022)
29. Li, K., Woo, M., Jia, L.: On the generation of disassembly ground truth and the evaluation of disassemblers. In: Proceedings of the 2020 ACM Workshop on Forming an Ecosystem Around Software Transformation. p. 9–14. FEAST'20, Association for Computing Machinery, New York, NY, USA (2020). 10.1145/3411502.3418429
30. Ince, T., Hollingsworth, J.K.: Compiler help for binary manipulation tools. In: Euro-Par 2012: Parallel Processing Workshops. pp. 404–413. Springer Berlin Heidelberg, Berlin, Heidelberg (2013). 10.1007/978-3-642-36949-0_46
31. Buck, B., Hollingsworth, J.K.: An api for runtime code patching. Int. J. High Perform. Comput. Appl. **14**(4), 317–329 (Nov 2000). 10.1177/109434200001400404
32. Williams-King, D., Kobayashi, H., Williams-King, K., Patterson, G., Spano, F., Wu, Y.J., Yang, J., Kemerlis, V.P.: Egalito: Layout-agnostic binary recompilation.

In: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems. p. 133–147. ASPLOS '20, Association for Computing Machinery, New York, NY, USA (2020). 10.1145/3373376.3378470

33. Panchenko, M., Auler, R., Nell, B., Ottoni, G.: Bolt: a practical binary optimizer for data centers and beyond. In: Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization. p. 2–14. CGO 2019, IEEE Press (2019). 10.1109/CGO.2019.8661201
34. Erosa, A.M., Hendren, L.J.: Taming control flow: A structured approach to eliminating goto statements. In: Proceedings of 1994 IEEE International Conference on Computer Languages (ICCL'94). pp. 229–240. IEEE (1994). 10.1109/iccl.1994.288377
35. Brumley, D., Lee, J., Schwartz, E.J., Woo, M.: Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In: 22nd USENIX Security Symposium (USENIX Security 13). pp. 353–368 (2013). 10.5555/2534766.2534797
36. Basque, Z.L., Bajaj, A.P., Gibbs, W., O'Kain, J., Miao, D., Bao, T., Doupé, A., Shoshitaishvili, Y., Wang, R.: Ahoy SAILR! there is no need to DREAM of C: A Compiler-Aware structuring algorithm for binary decompilation. In: 33rd USENIX Security Symposium (USENIX Security 24). pp. 361–378 (2024). 10.5555/3698900.3698921
37. Harel, D.: On folk theorems. Commun. ACM **23**(7), 379–389 (Jul 1980). 10.1145/358886.358892
38. Lee, J., Avgerinos, T., Brumley, D.: TIE: Principled reverse engineering of types in binary programs. In: 18th Annual Network and Distributed System Security Symposium (NDSS 2011) (2011)
39. Noonan, M., Loginov, A., Cok, D.: Polymorphic type inference for machine code. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 27–41 (2016). 10.1145/2908080.2908119
40. Lin, Z., Li, J., Li, B., Ma, H., Gao, D., Ma, J.: Typesqueezer: When static recovery of function signatures for binary executables meets dynamic analysis. In: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security. p. 2725–2739. CCS '23, Association for Computing Machinery, New York, NY, USA (2023). 10.1145/3576915.3623214
41. Xie, D., Zhang, Z., Jiang, N., Xu, X., Tan, L., Zhang, X.: Resym: Harnessing llms to recover variable and data structure symbols from stripped binaries. In: Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security. pp. 4554–4568 (2024). 10.1145/3658644.3670340
42. Di Luna, G.A., Italiano, D., Massarelli, L., Österlund, S., Giuffrida, C., Querzoni, L.: Who's debugging the debuggers? exposing debug information bugs in optimized binaries. In: Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. p. 1034–1045. ASPLOS '21, Association for Computing Machinery, New York, NY, USA (2021). 10.1145/3445814.3446695
43. Assaiante, C., D'Elia, D.C., Di Luna, G.A., Querzoni, L.: Where did my variable go? poking holes in incomplete debug information. In: Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2. p. 935–947. ASPLOS 2023, Association for Computing Machinery, New York, NY, USA (2023). 10.1145/3575693.3575720

44. Li, Y., Ding, S., Zhang, Q., Italiano, D.: Debug information validation for optimized code. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 1052–1065. PLDI 2020, Association for Computing Machinery, New York, NY, USA (2020). 10.1145/3385412.3386020