Ground Truth for Binary Disassembly is Not Easy

Chengbin Pang and Tiantai Zhang, *Nanjing University;* Ruotong Yu, *University of Utah;* Bing Mao, *Nanjing University;* Jun Xu, *University of Utah*

https://www.usenix.org/conference/usenixsecurity22/presentation/pang-chengbin

This paper is included in the Proceedings of the 31st USENIX Security Symposium.

August 10–12, 2022 • Boston, MA, USA

978-1-939133-31-1

Ground Truth for Binary Disassembly is Not Easy

Chengbin Pang[†] *Tiantai Zhang*[†] *Ruotong Yu*[‡] *Bing Mao*[†] *Jun Xu*[‡]
[†]*State Key Laboratory for Novel Software Technology, Nanjing University*
[‡]*School of Computing, University of Utah*

Abstract

Modern disassembly tools often rely on empirical evaluations to validate their performance and discover their limitations, thus promoting long-term evolvement. To support the empirical evaluation, a foundation is the right approach to collect the ground truth knowledge. However, there has been no unanimous agreement on the approach we should use. Most users pick an approach based on their experience or will, regardless of the properties that the approach presents.

In this paper, we perform a study on the approaches to building the ground truth for binary disassembly, aiming to shed light on the right way for the future. We first provide a taxonomy of the approaches used by past research, which unveils five major mechanisms behind those approaches. Following the taxonomy, we summarize the properties of the five mechanisms from two perspectives: (i) the coverage and precision of the ground truth produced by the mechanisms and (ii) the applicable scope of the mechanisms (e.g., what disassembly tasks and what types of binaries are supported). The summarization, accompanied by quantitative evaluations, illustrates that many mechanisms are ill-suited to support the generation of disassembly ground truth. The mechanism best serving today's need is to trace the compiling process of the target binaries to collect the ground truth information.

Observing that the existing tool to trace the compiling process can still miss ground truth results and can only handle x86/x64 binaries, we extend the tool to avoid overlooking those results and support ARM32/AArch64/MIPS32/MIPS64 binaries. We envision that our extension will make the tool a better foundation to enable universal, standard ground truth for binary disassembly.

1 Introduction

Disassembly is the process of reversing basic constructs, such as instructions and functions, from binary code. It offers the foundation for binary analysis and the downstream security applications (e.g., code layout randomization [12,15], control flow integrity [31,37], and similarity measurement [11,16,36]). To support disassembly, an abundance of tools have been created, ranging from open source ones (e.g., ANGR [33] and GHIDRA [3]) to commercial ones (e.g., IDA PRO [13] and BINARY NINJA [25]). To the success of these tools, a critical but easily overlooked factor is the generation of accurate ground truth of disassembly results. On the one hand, ground truth knowledge is a foundation to build many tools. In particular, data-mining or machine learning based tools [10,28,32,35] need ground truth to label data for training the models. On the other hand, ground truth information is indispensable to measure the disassembly outcomes, which drives the evolvement of nearly every tool.

Despite the importance of ground truth for disassembly, it has not received due attention. Past research on binary disassembly has been creating/obtaining ground truth somewhat arbitrarily, without sufficient consideration of the rigor. Take the recovery of instructions as an example. As far as we know, there are four distinct strategies to obtain the ground truth. Meng *et al.* [22] run manual analysis to obtain the ground truth, while ZAFL [23] simply considers the results produced by existing disassemblers (OBJDUMP) as the ground truth (or precisely, baseline) for comparison. In contrast, XDA [28] and Andriesse *et al.* [5] combine instruction locations embedded in the debug information and linear sweeping to collect legitimate instructions. More intelligently, Pang *et al.* [26] trace the compiling, assembling, and linking process to gather all the instructions emitted by the compilation toolchain. However, little has been done to inspect the fidelity of various types of ground truth and the implications behind. Instead, the community seems to have been permissive towards the choice of ground truth and accept whatever being used.

In this paper, we focus on the above concern about ground truth for binary disassembly. We start with a taxonomy of the approaches to building ground truth information for major disassembly tasks (recovery of instructions, function boundaries, and control flows). The taxonomy describes the internal principles and mechanisms of each approach. Overall, five different mechanisms are being used nowadays. These mechanisms are heterogeneous in nature, ranging from labor-intensive ones (① *manual analysis*) to opportunistic ones (② *reusing existing disassemblers*) and compiler-aided ones (③ *leveraging compilation metadata*, ④ *exploiting intermediate compiler outputs*, and ⑤ *tracing the compiling process*).

Following the taxonomy, we run a qualitative analysis to compare the five mechanisms from two key perspectives that affect their applications, including (i) the recall and precision of the ground truth produced by the mechanisms and (ii) the applicable scope of the mechanisms (e.g., what disassembly tasks and what types of binaries are supported). It turns out that many of the mechanisms are somewhat ill-suited to support the generation of ground truth. Most notably, they lack the necessary foundations to ensure coverage and correctness, which tend to present inadequate recall and precision. Throughout further, quantitative evaluations, we validate that the lack of recall and precision can very often lead to incomplete observations and unreliable conclusions. For instance, Meng *et al.* [22] rely on manual analysis to collect a small set of ground truth results for evaluating their DYNINST tool, which demonstrates full accuracy of DYNINST in handling complex constructs. However, extending the ground truth to include all the results, we observe that DYNINST may not offer perfect accuracy.

While our study and evaluation unveil the inappropriateness of many existing approach, we, fortunately, identify that the mechanism of *tracing the compiling process* can largely meet the requirement of providing ground truth for binary disassembly. The mechanism essentially reports the information that the compiler considers as the ground truth when compiling the target binaries. In result, this mechanism offers guaranteed precision and supports all kinds of disassembly tasks. However, only one tool, developed by Pang *et al.* [26], supports tracing the compiling process to collect disassembly ground truth. And the tool has two major issues to serve today's needs: it still misses a set of ground truth that the compiler cannot recognize and it can only handle x86/x64 binaries. Motivated by the potential of the tool, we extend it to re-collect the missing ground truth and to support ARM32/AArch64/MIPS32/MIPS64 binaries. Applying the extended tool to re-evaluate mainstream disassembly tools, we derive a group of previously less-known findings. For instance, commercial tools, such as IDA PRO and BINARY NINJA, present substantially downgraded performance when disassembling MIPS binaries. We anticipate that our extension will make the tool a better foundation to enable universal, standard ground truth for binary disassembly.

In summary, we make the following main contributions.

- We present a systematic taxonomy of the approaches and mechanisms to collect ground truth for binary disassembly. The taxonomy brings a better view of what are being used and accepted today.
- We perform a qualitative analysis to unveil the limitations of existing approaches for disassembly ground truth. We further conduct a quantitative evaluation to demonstrate the implications and harms those limitations can bring to the applications. The analysis and evaluation provide evidence for the need of better, trustworthy approaches.
- We extend the state-of-the-art tool to enable complete, precise, and widely applicable collection of ground truth for binary disassembly. This piece of extension helps pave the way to standardize and unify the ground truth for binary disassembly evaluation. The code and dataset are available at https://github.com/junxzm1990/x86-sok.

2 Taxonomy of Ground Truth Approaches

There has been no agreement on the approach to creating the ground truth for binary disassembly. Thus, various approaches exist and are being used. To systematically understand those approaches, we first categorize them based on their internal mechanisms. In general, the approach for ground truth is tied to the disassembly task. To better bound our research, we focus on ground truth for three major disassembly tasks (definitions are adapted from previous research [10, 22, 26, 27]).

- **Instruction Recovery** is the process of identifying instructions emitted by the compiler or introduced by the developer in a binary program.
- **Function Detection** is to reconstruct the mapping from the code in a binary to the corresponding functions in the source code. In this paper, we focus on the detection of function starts, considering that other function information can be easily obtained by combining the function start and the control flows [6].
- **CFG Reconstruction** re-builds the control flow graph (CFG) of a binary program. This paper only discusses control flows in the form of indirect jumps. Other control flows are either trivial to obtain (e.g., direct jumps/calls) or less feasible to reconstruct (e.g., indirect calls), and modern disassemblers do not particularly handle them.

On the market, the existing approaches to obtaining ground truth prevalently adopt five mechanisms. They take the target binaries as inputs and output constructs required by the above disassembly tasks as the ground truth. They also often assume they have access to the compilation process of the target binaries.

Running Manual Analysis. This is an intuitive but less popular approach. Meng *et al.* [22] used this approach when evaluating their DYNINST tool. They manually collected the ground truth for a small set of instructions, functions, and control flows. In particular, they focused on challenging code constructs, such as non-instruction bytes in code, non-contiguous functions, and jump tables.

Reusing Existing Disassemblers. From time to time, people consider existing, reputed disassemblers as the oracle and use their outputs as the baseline for comparison. Nagy *et al.* [23] run LLVM-OBJDUMP on target binaries to get the baseline of instructions, when measuring the coverage of their ZAFL tool. In contrast, Kinder *et al.* [18] take jump tables detected by IDA PRO as the baseline to evaluate their JAKSTAB tool in reconstructing control flows.

Exploiting Intermediate Compiler Outputs. Modern compilers can be configured to output certain intermediate results, which have been exploited by recent research [21,34] to derive disassembly ground truth.

```
offsets raw bytes  instructions
0000:   F30F1EFA    endbr64
0004:   55          push %rbp
0005:   4889E5      mov  %rsp, %rbp
0008:   488D3D00    lea  .LC0(%rip), %rdi
```

Listing 1: A snippet of listing files by GAS Assembler.

Li *et al.* [21] leverage the listing files produced by assemblers to obtain the ground truth of instructions. Listing 1 presents an example of listing files produced by GNU Assembler when given the option of `-listing-rhs-width=1024`. A listing file is tied to an object file, which gives offsets of all instructions in each function belonging to the object file. Adding the offset to the start address of a function recorded by its symbol in the linked binary, one can calculate the final address of each instruction in the function.

```
;; Function main (main, funcdef_no=9, decl_uid=2839)
...
(jump_insn # (parallel [(set (pc) (reg:DI 0 ax [92]))
 ])# {*tablejump_1} ; this is a jump table reference)
(jump_table_data # 0 0 (addr_diff_vec:SI
     (label_ref:DI #)
        [ ; array of jump table entries
           (label_ref:DI #)
           (label_ref:DI #)
           (label_ref:DI #)
           (label_ref:DI #)
           (label_ref:DI #) ]
```

Listing 2: Example of RTL output by GCC. It shows that the `main` function contains one jump table with five targets.

David *et al.* [34] configure GCC to dump the final internal representation (RTL) during compilation, using the developer option `-fdump-final-insns`. They then extract "rough" ground truth about jump tables from the RTL file. As shown in Listing 2, RTL unveils the number of jump tables and the number of targets of each jump table in each function, which is deemed as the ground truth in [34].

Leveraging Compilation Metadata. The compilation tools, given the needed options, can maintain various metadata in the produced binaries. Such metadata, including symbols and debug information (`-g`), is often used to obtain disassembly ground truth.

Using addresses embedded in symbols as the ground truth of function starts is a de facto standard strategy [5, 6, 10, 28, 35]. This piece of ground truth has also been used for different goals. Andriess *et al.* [5] and Nucleus [6] leverage symbols to measure their disassemblers' performance of function detection. Byteweight [10], XDA [28], and DEEPDI [35] apply symbols to train and test their machine learning models for function detection.

ARM binaries can also carry a special type of symbols called mapping symbols [8], which mark the beginning address of a sequence of ARM code, Thumb code, or data. Jiang *et al.* [17] exploit the mapping symbols to collect ground truth of instructions. Their idea is to linearly disassemble the instruction from the beginning of a ARM/Thumb code region until the next mapping symbol. In fact, OBJDUMP works in a similar way when such mapping symbols exist. In this regard, Jiang *et al.* [17] is essentially "reusing" OBJDUMP to obtain ground truth of instructions.

Unlike symbols, debug information is mostly used to produce the ground truth of instructions and jump tables. Andriess *et al.* [5] and XDA [28] collect addresses of legitimate instructions encoded in the line information and linear sweep the regions between any two addresses to build ground truth of instructions. For conservativeness, they stop the linear sweeping when encountering a control flow instruction. Andriess *et al.* [5] further leverage the line information to pinpoint jump tables. Specifically, they map `switch` statements and their cases in the source code to indirect jumps and their targets in the binary, based on the line information.

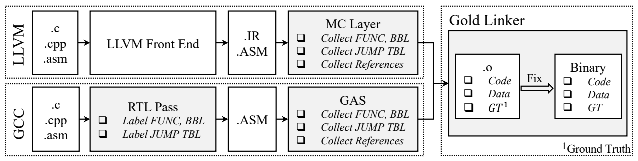


Figure 1: The tools Pang *et al.* [26] use to obtain ground truth for binary analysis. The boxes with grey color indicate components that are modified in LLVM, GCC, and Gold Linker.

Tracing Compiling Process. The last mechanism is proposed by Pang *et al.* [26]. They trace the end-to-end compiling, assembling, and linking procedure to collect various types of ground truth. Their approach follows the idea of CCR [19]. As shown in Figure 1, CCR extends the LLVM Machine Code (MC) layer. While assembling a bitcode file or an assembly file to an object file, the extended MC layer collects information about basic blocks, functions, and jump tables, and keeps the information in an extra section. To merge information from multiple object files, CCR further instruments the GNU gold linker to adjust the above items in the process of linking.

Pang *et al.* also port the idea of CCR to GNU GCC. By instrumenting the RTL pass in GCC, they insert primitives to label functions, basic block, and jump tables in the assemble code. In the procedure of assembling, these primitives facilitate the customized GNU Assember (GAS) to collect information about the corresponding items. Similar to CCR, they save the collected information as a new section in each object file and reuse the CCR linker for merging the object files.

3 Properties of Ground Truth Approaches

The diverse group of ground truth approaches offer a wide range of choices, but there lacks a systematic comparison of those approaches. In this section, we inspect the existing approaches from four perspectives that affect their applications.

- **Precision** describes the ratio of correct results in the "ground truth" reported by an approach. Formally, $precision = \frac{|TP|}{|TP|+|FP|}$ (TP and FP stand for true positives and false positives). A low precision means the ground truth includes many errors and should not be trusted.

Table 1: Qualitative comparison of existing approaches to building ground truth for binary disassembly. The degree of filling in a circle shows how well a mechanism satisfies a property. For instance, ● means full satisfaction and ○ indicates zero satisfaction.

Mechanisms	**Applications**	**Properties**			
		`Precision`	`Recall`	`Generality`	`Extendibility`
Running Manual Analysis	[22]	●	◔	●	●
Reusing Existing Disassemblers	[18, 23]	◕	◕	●	●
Exploiting Intermediate Compiler Outputs	[21, 34]	●	◕	◔	◔
Leveraging Compilation Metadata	[5, 10, 17]	◕	◕	◔	●
Tracing compilation Process	[26]	●	◕	●	◔

- **Recall** measures the ratio of all correct results covered by the reported ground truth. Formally, $recall = \frac{|TP|}{|TP|+|FN|}$ (FN stands for false negatives). Ground truth with limited coverage may also not be trusted as it gives incomplete information, which can be biased and misleading
- **Generality** requires that ground truth approach supports various disassembly tasks. The common ones include instruction recovery, function detection, and CFG reconstruction, as described in section 2.
- **Extendibility** concerns the types of binaries that the ground truth approach can be applied to. Two common metrics of extendibility are *what architectures are supported* (e.g., x86, ARM, and MIPS) and *what compilers are supported* (e.g., GCC, Clang, and MSVC).

Table 1 summarizes the above properties of the approaches we categorized in section 2.

Running Manual Analysis. Assuming the analysts are experienced and cautious, this approach shall offer ground truth with extremely high precision. In addition, the approach can be generally applied to any disassembly tasks and binaries produced by any compilers/running on any architectures. This implies high generality and high extendibility.

```
;original code                      ; replaced version
vpmull.p64    $Xl,$H,$H             .word   0xf2a02ea0
...                                 ...
vpmull2.p64   $Xh,$H,$H             .word   0xf2a94ea9
vpmull.p64    $Xm,$t0,$t0           .word   0xf2a02ea0
```

Listing 3: Code labeled as data in `openssl1.1.0l`, ARM version. Before assembling, `vpmull.p64/vpmull2.p64` instructions (*left*) are replaced by `.word` bytes (*right*) to accommodate assemblers that cannot recognize those instructions.

The primary drawback of the approach is its limited scalability. Given benchmarks with larger binaries, it is practically infeasible to gather all the ground truth manually. Thus, the approach tends to present a limited recall. This is also reflected by that Meng *et al.* [22] only picked ten to twenty instances of each construct. Despite the overall low coverage, manual analysis (precisely, human intelligence) is indispensable to cover certain results. Listing 3 shows an example where code is labeled as `.words` in the assembly file. The assembler will emit them as data in the code region. In this case, human intelligence is needed to understand the replacing procedure and thus, identify the instructions.

Reusing Existing Disassemblers. This is the most straightforward approach to obtaining the ground truth. It supports all disassembly tasks and all binaries that the underlying disassemblers can handle. For instance, using IDA PRO for ground truth will cover nearly everything, thus presenting perfect generality and extendibility.

```
1 fgetspent_r:
2 1003ab: je 1003ae <fgetspent_r+0x3e>
3 1003ad: lock cmpxchg %ecx,(%ebx)
4 1003ae: cmpxchg %ecx, (%edx) ; overlapped instruction
```

Listing 4: Overlapped instructions in `glibc` compiled by GCC-8.1 with O2. Line 3 and line 4 are two overlapped instructions that share the same part starting at `0x1003ae`. OBJDUMP can only recognize the instruction at `0x1003ad`.

In principle, using this approach means considering the underlying disassembler as the oracle. Unfortunately, none of the existing disassemblers provides perfect recall or precision, regardless of the disassembly tasks. Consider the case of using OBJDUMP [23] for baseline of instructions as an example. OBJDUMP has the well-known drawback of misrecognizing data as code, often creating false positives in the recovered instructions. Even only considering the recall of OBJDUMP like Nagy *et al.* [23], it has the issue of skipping overlapped instructions. Listing 4 shows such an example.

The situation of using IDA PRO for baseline of jump tables is similar. As we will show in section 4, IDA PRO produces hundreds of false positives and false negatives when running on the x86/x64 benchmarks presented in [26]. Applied to MISP benchmarks, the results are even more concerning. Both false positive rate and false negative rate are significantly higher. Related details are covered in subsection 6.4.

Exploiting Intermediate Compiler Outputs. Ground truth obtained with this approach is essentially compilation result that will appear in the binary. For instance, listing files include instructions eventually constitute the code in the binary. Assuming no optimizations happen after generation of the intermediate outputs (e.g. no link time optimizations), the fidelity (or precision) of the ground truth is guaranteed. In most cases, the recall of the ground truth is also guaranteed. The only exceptions happen when the compiler does not know the ground truth (e.g. Listing 3) or it does not explicitly output the ground truth in the intermediate results (e.g. Listing 4).

Despite the guaranteed precision and recall, ground truth extracted from intermediate compiler outputs is often not easy to use. Specifically, the results are labeled with intermediate identities, making it complicated to map them to their counterparts in the final binary. In the case of using listing files to collect instructions [21], the available information is function names and offsets of instructions in each function. To find those instructions in the final binary, symbols are then used to map function names to their addresses in the binary program. This method works in most cases but can fail when aliased functions exist[1]. To further distinguish aliased functions, Li *et al.* leverage the debug information to uniquely associate each function with its source code location. Doing so helps but inevitably increases the complexity and inherits the defects of debug information. Using RTL for jump tables has similar issues. RTL can only tell the number of jump tables and the number of targets of each jump table in a function, which cannot map to the specific instances in the binary. Applying such ground truth, for example, to evaluate the performance of disassemblers, cannot pinpoint the errors when detected and can even mask real errors.

Another major issue of this approach is the limited generality and extendibility. It fully depends on what outputs the compiler is designed to export. For instance, listing files can only be used to collect instructions and RTL may only help jump tables. There lacks similar outputs for other disassembly tasks. In addition, listing files is only supported by GNU assembler in Linux and RTL is a unique output of GCC. Such intermediate results may not be available when using a different compilation toolchain.

```
__bn_postx4x_internal:
  mov   8*0($nptr),%r12
  mov   %rcx,%r10     # -$num
  ...
call  __bn_postx4x_internal
```

Listing 5: Handwritten assembly code in `openssl1.1.0l`. Line 1 defines a function but does not label it with `.type @function`. No code symbol is created for the function.

Leveraging Compilation Metadata. This approach has a principle similar to exploiting intermediate compiler outputs if considering compilation metadata as compiler outputs. There is, however, a fundamental difference. Intermediate compiler outputs are essentially the ground truth before converted to the final form in the binary, while compilation metadata is auxiliary data that happens to carry ground truth information for disassembly. This difference affects both precision and recall of the ground truth.

```
; line information
Address     Line
0x4402cb    334
0x4402da    334
```

```
4402cb: call 0x403760
4402d0: mov 0x40(%rsp),%r8
4402d5: mov 0x30(%rsp),%r9
4402da: mov  (%rax),%rdx
```

Listing 6: Example in `findutils` where the approach using debugging information [5] misses legitimate instructions. The *left* part shows two continuous records of line information and the *right* part shows the related instructions. The approach in [5] starts linear disassembly at the address encoded in the first line information (`0x4402cb`) and stops at that instruction as it transfers the control flow. It then continues the disassembly at the address encoded in the second line information (`0x4402cb`). This way, it misses two instructions at `0x4402d0` and `0x4402d5`.

Symbols are created to facilitate symbolization of code or data in scenarios like linking and debugging. A code symbol, having the `STT_FUNC` type and representing a contiguous code region, mostly corresponds to a function but it is not mandated. Modern compilers like GCC can split a function into discontinuous regions (e.g., hot/cold function splitting [20]) and attaches a separate symbol for each region to accommodate debugging. Thus, using code symbols as the ground truth of function starts can introduce false positives when discontinuous functions appear. In addition, the developers may occasionally omit the `@function` type for function names when creating handwritten assembly code. The compiler will not introduce a code symbol for these functions. Thus, false negatives can also arise when using symbols as the ground truth for functions. Listing 5 presents such as an example.

```
; debug line
Address     Line
0x493b17    23
0x493b1f    23
```

```
493b17: mov   %rax,%rbp
493b1a: jmpq  46de4e
493b1f: 00 ; alignments
493b20: push  %rbx
```

Listing 7: Flase positive of debug information in `filezilla` compiled by GCC-8.1. The left part is two continuous records of debug information. It has the record of address `0x493b1f`. While `0x493b1f` is a one byte alignment as shown in right part.

Debugging information is even more problematic when leveraged to collect disassembly ground truth. It typically includes line information to keep the address of the first instruction compiled from each intermediate representation statement (e.g., GCC GIMPLE). Thus, the line information encodes locations of many instructions but not all of them. As pointed out in section 2, attempts have been made by combing the line information and conservative disassembly to gather

[1]Compilers can link aliased functions in the same binary if the functions are declared as weak symbols (e.g., `__attribute__(weak))` for GCC

ground truth of instructions. Not surprisingly, this approach is not perfect. According to our preliminary evaluation (as shown in Appendix A), it may miss millions of legitimate instruction when applied to the x86 benchmarks presented in [26]. Listing 6 illustrates why with an example. This approach can also introduce many false instructions as line information may point to alignment bytes instead of real instructions, as shown in Listing 7. These problems of line information have similar impacts when applied on mapping switch-cases in source code to jump tables in the binary.

While compilation metadata brings less complete and less precise ground truth, it still has advantages compared to intermediate compiler outputs. First, compilation metadata is easier to use because it carries final information in the binary (e.g., final address of an instruction) instead of intermediate results. Second, compilation metadata, including symbols and debugging information, is a standard feature of modern compilers, regardless of the compiler version and the architecture the produced binaries will run on. Hence, relying on compilation metadata to produce disassembly ground truth shall have excellent extendibility.

Tracing Compiling Process. The key idea of this approach is to track the steps where the compiler generates the constructs needed by the ground truth. It tags those constructs all the way until they arrive at the final binaries. In a general sense, the constructs are what the compiler views as the ground truth. Thus, their precision is guaranteed and will not incur problems that other approaches have. For instance, the approach records all the basic blocks that the compiler creates for a function, regardless of where the basic blocks are placed in the binary. This way, it can recognize non-contiguous functions and avoids false positives that symbol-based approaches produce.

Similar to using intermediate compiler outputs, this approach offers high recall but cannot cover the cases that the compiler fails to recognize. Listing 3 and Listing 5 show such examples of instructions and functions. In addition, developers may often create handwritten jump tables in assembly files, which the compiler can also miss (see subsection 6.2). Listing 4 demonstrates another special class of cases. In this example, the compiler generates a `jump` from `0x1003ab` to `0x1003ae`, showing its awareness of the instruction at `0x1003ae`. Capturing this case requires fine-grained tracing of compiler operations on generating assembly code, which the tool proposed by Pang *et al.* [26] does yet not support.

Besides guaranteed precision and extremely high recall, this approach has two other advantages. First, it can support nearly every disassembly task because it can freely collect information from the compilation process. In this regard, the approach has unrestricted generality. Second, the results it collects are what present in the final binary. No extra mapping or processing is needed before using the results.

The major drawback of this approach is the need for customizing the compiler, the assembler, and the linker. To support a new compiler or a new architecture, high domain knowledge and heavy engineering efforts are required. In this regard, the approach has limited extendibility.

4 Implications of Imperfect Ground Truth

The existing approaches to obtaining disassembly ground truth have varying properties. In this section, we seek to shed light on how the properties affect the applications of those approaches. In particular, we focus on *precision* and *recall*. Generality and extendibility also matter but their impact is unanimously on whether the approach can be used or not.

To support the understanding, we consider the approach by tracing the compiling process as the *oracle approach* and use its results as the *"golden" ground truth*. Specifically, we patch the tool developed by Pang *et al.* [26] to mitigate the recall issues discussed above and run the patched tool to collect the ground truth for instructions, function starts, and jump tables. Technical details of our patch are shortly presented in section 6. We also recompile the benchmarks presented at `https://github.com/junxzm1990/x86-sok` to work as the target binaries. Following the setup of [26], all target binaries are compiled to run on the x86/x64 architectures. We omit other architectures because the related tools either only support x86/x64 or the related results will be discussed in subsection 6.4.

Table 2: Performance of BYTEWEIGHT trained with ground truth produced by different approaches (`Symbol` indicates using symbols as the ground truth of function starts while `Oracle` indicates using the approach presented in [26].

Metric	GT	O0	O1	O2	O3	Os	Of
Recall	`Symbol`	99.52	96.94	98.01	98.38	96.00	98.39
	`Oracle`	99.54	96.94	97.66	98.18	96.19	98.18
Precision	`Symbol`	99.67	97.42	94.73	90.88	98.35	90.61
	`Oracle`	99.69	97.89	99.21	99.37	98.46	99.39
F1-Score	`Symbol`	99.59	97.17	96.34	94.48	97.18	94.33
	`Oracle`	99.61	97.41	98.43	98.77	97.57	98.78

Impacts on Training Accuracy. Disassemblers based on data mining or machine learning need ground truth knowledge when training the classification models. Intuition suggests that fidelity of the ground truth can affect the model accuracy. To this end, we perform a study on BYTEWEIGHT [10], a data mining based approach to detecting function starts. We train BYTEWEIGHT twice, respectively using ground truth obtained from symbols and produced by the oracle approach [26]. Symbols, instead of other approaches, are considered as the baseline to align with the setting of the original evaluation on BYTEWEIGHT [10]. Both BYTEWEIGHT models are then tested using the ground truth offered by the oracle approach to understand their precision, recall, and accuracy (F1-score: $\frac{2*Precision*Recall}{Precision+Recall}$). All the training and testing are based on `coreutils-8.30` compiled with `GCC-8.1` (no overlap between the training dataset and the testing dataset). To diversify the evaluation, common optimization levels (O0, O1, O2, O3, Os, Ofast) are all considered.

Table 2 summarizes the evaluation results. Evidently, the choice of ground truth has an impact on the model accuracy, regardless of the optimization level. Specifically, models trained with the golden ground truth present a consistently higher accuracy. At higher optimization levels (O3 and Of), the accuracy difference can exceed 4%. These results are anticipated considering that symbols carry many false function starts introduced by non-contiguous functions.

Table 3: Performance of XDA on instruction recovery trained with ground truth produced by different approaches (`Debug` indicates using the debug information based approach proposed in [5] while `Oracle` indicates using the approach presented in [26] (with our patch presented in subsection 6.2).

Metric	GT	O0	O1	O2	O3	Os	Of
Recall	`Debug`	91.47	91.82	91.91	91.66	92.61	91.38
	`Oracle`	96.76	95.37	95.38	95.11	95.13	94.89
Precision	`Debug`	99.21	98.92	98.84	99.24	96.05	99.29
	`Oracle`	99.26	98.96	98.89	99.27	95.07	99.31
F1-Score	`Debug`	95.18	95.23	95.25	95.30	94.30	95.17
	`Oracle`	97.99	97.13	97.10	97.14	95.15	97.05

We further extend the experiment on running XDA [28], a transfer learning based disassembler, for instruction recovery. In this experiment, two non-overlapped subsets of `coreutils-8.30` and `findutils-4.4` are respectively picked for training and testing. We further use two different ground truth approaches, the oracle approach [26] and the debug information based approach [5], to label the training data. In contrast, the testing data is unanimously labeled using the oracle approach. The experiment results are summarized in Table 3. Again, the ground truth approach has an observable impact on XDA. In particular, the recall of XDA can drop by 4% when using an improper approach of ground truth.

Finding #1: The fidelity of ground truth affects the utility of disassembly tools built with data mining or machine learning. Highly accurate ground truth is desired.

Impacts on Tool Evaluation. The testing of disassembly tools, which is highly critical to their evolvement, greatly depend on ground truth knowledge. The recall and precision of the ground truth can both affect the evaluation. In particular, incomplete or inaccurate ground truth can provide distorted evidence towards unreliable conclusions. We elaborate on three such cases in the following.

Case 1: Meng *et al.* [22] leverage manual analysis to collect ground truth to measure their tool, DYNINST, in identifying complex constructs. As we pointed out before, they only covered a small set of the ground truth (10-20 instances of each construct) due to the heavy burden. Using this set of ground truth, they obtained the observation that DYNINST achieves full recall and full precision. We extended the benchmark to include all the x86/x64 binaries presented in [26] and run the oracle approach (with our patch) to gather the ground truth of three complex constructs aligned with our target disassembly tasks (data embedded in code, jump tables, and tail calls[2]). Table 4 presents the evaluation results using our extended benchmark. Evidently, DYNINST may not provide perfect recall and precision, invalidating the observation presented in [22]. This case demonstrates that incomplete ground truth can lead to biased claims and conclusions.

Table 4: Performance of DYNINST on identifying complex constructs under different approaches to extracting the ground truth. `Manual` indicates the ground truth is manually collected from the testsuite presented in [22]. `Oracle` indicates the ground truth from the testsuite presented by [26]. **Embedded** means data embedded in code, which is only evaluated on binaries that contain data-in-code.

Metric	GT	Embeded	JMPTBL	Tail Call
Recall	`Manual`	100.0	100.0	100.0
	`Oracle`	89.55	98.61	71.7
Precision	`Manual`	100.0	100.0	100.0
	`Oracle`	99.35	99.83	67.39
F1-Score	`Manual`	100.0	100.0	100.0
	`Oracle`	94.19	99.21	69.48

Table 5: Performance of ZAFL on instruction recovery. `Objdump-Sym` indicates we run OBJDUMP on binaries with symbols for the ground truth while `Objdump` means we run OBJDUMP on stripped binaries. `Oracle` shows the results of using the approach presented in [26] to get the ground truth.

Metric	GT	O0	O2	O3	Os	Of
# of FP	`Objdump-Sym`	0	3	3	135	134
	`Objdump`	0	3	3	135	134
	`Oracle`	0	17	40	203	189
# of FN	`Objdump-Sym`	628.8K	754.5K	733.4K	354.6K	860.8K
	`Objdump`	683.8K	819.4K	797.8K	408.4K	928.2K
	`Oracle`	83.4K	84.1K	84.2K	145.7K	145.1K

Case 2: Nagy *et al.* [23] consider the results of OBJDUMP as the baseline to measure their ZAFL tool in recovering instructions. They observed that ZAFL incurs zero false positives on both closed-source and open-source binaries. They also reported that ZAFL misses no instructions that 24-hour fuzzing can reach. We reproduced the evaluation with three approaches to generate the ground truth: (i) OBJDUMP with symbols available in the target binaries; (ii) OBJDUMP with stripped binaries; and (iii) the oracle approach with our patch. We also used the x86/x64 benchmarks developed by Pang *et al.* [26] except for those compiled by Clang for 32-bit machines[3]. The evaluation results are presented in Table 5.

[2]Identification of tail calls can be trivially done when the results of instructions, functions, and jump tables are available

[3]ZAFL relies on the information carried in `.eh_frame` to determine the

Not surprisingly, the ground truth approach affects the observations. Using the oracle approach, we observe more false positives in most cases. This indicates ZAFL may make more mistakes than it was believed to. On the other hand, ZAFL seems to produce much fewer false negatives than what the OBJDUMP-based measurement shows. It is worth mentioning that the false negatives we observe do not conflict with the conclusions in [23], as we do not measure how many of the false negatives can be reached by fuzzing in 24 hours.

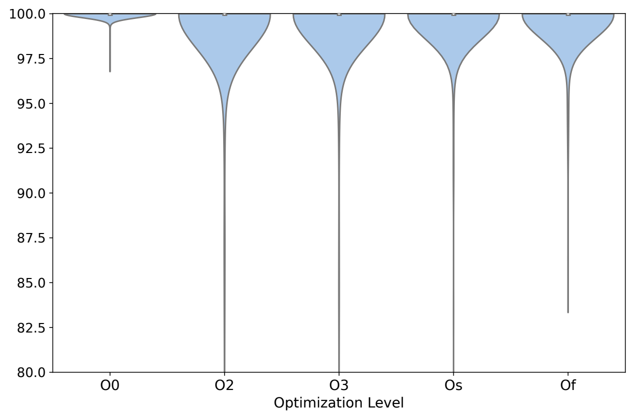


Figure 2: Distribution of precision of IDA PRO on detecting jump tables in the x86/x64 binaries presented in [26].

Case 3: Kinder *et al.* [18] consider the results of IDA PRO as the baseline for evaluating their JAKSTAB tool in detecting jump table related constructs. A key observation they obtained is that JAKSTAB produces nearly zero false positives. We hoped to re-evaluate JAKSTAB using ground truth produced by different approaches. However, JAKSTAB cannot run modern benchmark binaries due to a parsing error (https://github.com/jkinder/jakstab/issues/9), which was also observed by Pang *et al.* [26].

Alternatively, we chose to measure the precision of IDA PRO in detecting jump tables, considering the ground truth produced by the oracle approach as the baseline. In the measurement, the benchmark consists of all x86/x64 binaries presented in [26]. Figure 2 shows the distribution of the precision of IDA PRO on different binaries. In a nutshell, IDA PRO is not perfectly precise. It makes many mistakes, in particular at the higher optimization levels. We envision this imprecision in the ground truth used by Kinder *et al.* may pose a risk on their observations and understandings of JAKSTAB. However, we cannot directly confirm this as we did not run JAKSTAB.

Finding #2: The use of incomplete or imprecise ground truth can lead to misleading observations and conclusions about the disassembly tools, which can impede their future development.

ranges of disassembly. However, the 32-bit binaries produced by Clang do not contain `.eh_frame`.

Impacts on Tool Comparison. In recent years, there have been a series of studies on comparing different disassembly tools [5, 17, 26]. This is another scenario where ground truth information is indispensable. A common goal of the studies is to rank the tools based on their performance. Hypothetically, the ranking can vary when different ground truth approaches are used.

We perform an empirical study to experiment on the hypothesis. A challenge of the study is the selection of benchmark binaries. Many benchmarks only contain easy constructs of the ground truth, which can be perfectly identified by every ground truth approach. Using those benchmarks makes little sense for our study. To address this issue, we revisited the benchmarks presented in [26] and finally picked `Openssl-1.1.0l`. The primary reason is `Openssl-1.1.0l` carries plenty of complex constructs (e.g., handwritten assembly and data in codes), which the existing ground truth approaches cannot handle well.

We compiled `Openssl-1.1.0l` with GCC-8.1 into x86/x64 binaries under different optimization levels (O0, O1, O2, O3, Os, and Ofast). We focus on x86/x64 architectures because the original evaluation of many mainstream disassemblers consider this architecture [5, 21, 26, 28]. Utilizing the binaries, we measure the accuracy (F-1 score) of 8 popular disassemblers on recovering instructions three times, respectively using ground truth produced by OBJDUMP with symbols, the debug information based approach proposed in [5], and the compiler-tracing approach proposed in [26] (after applying our patch presented in section 6). Figure 3 summarizes the results averaged on the optimization levels.

Evidently, the disassemblers present "varying" accuracy when the approach to obtaining the ground truth changes. This accuracy difference can often lead to variation of the rankings. Consider the comparison between IDA PRO and BINARY NINJA, the two state-of-the-art commercial disassemblers, as an example. BINARY NINJA outperforms IDA PRO when measured using the less accurate ground truth (`objdump`). However, the result flips when we switch to the oracle ground truth, despite the margin is small.

Finding #3: Different ground truth approaches “affect” the performance of disassembly tools. When used in evaluation studies, the ground truth approaches can make the rankings deviate from the reality.

5 Discussion: What Do We Need Today?

The previous sections bring us a better view of the existing approaches to generating ground truth for disassembly, including their mechanisms, properties, and implications. A key, follow-up question is *what approaches we need today*.

Recall and Precision. We argue that the most important criterion of selecting ground truth approaches shall be recall and

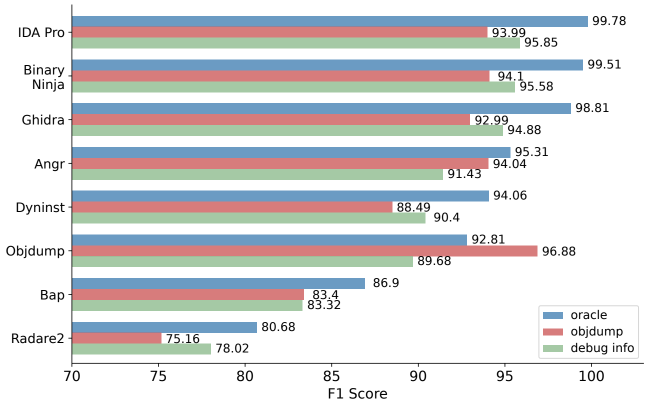


Figure 3: Accuracy (F-1 score) of popular disassemblers on recovering instructions, measured using different ground truth approaches (`oracle` indicates the compiler tracing approach presented in [26], `objdump` means using objdump with symbols, and `debug info` represents the debug information based approach proposed in [5]).

precision. On the one hand, recall and precision are scientific standards of ground truth in general. In other domains and scenarios, ground truth is expected to be complete and precise. On the other hand, our studies presented in section 4 unveil various scenarios where incomplete or imprecise ground truth leads to biased observations and improper conclusions.

Among the existing approaches to building the ground truth, manual analysis and reusing existing disassemblers lack necessary mechanisms to ensure recall or precision. Leveraging compilation metadata inherits the knowledge from the compiler, which seems to be more promising. However, the knowledge is often not exactly the ground truth. Conversions from the knowledge to the needed ground truth can introduce errors or miss cases. In this regard, leveraging compilation metadata also has an insufficient guarantee of recall and precision.

In contract, the remaining two approaches, exploiting intermediate compiler outputs and tracing the compiling process are better grounded. Their inputs are ground truth directly output by the compiler. Thus, they are guaranteed by the compiler to present complete and accurate results except for the few cases where the compiler has insufficient knowledge. From this perspective, we conclude that the two approaches are more desired today.

Generality and Extendibility. Modern disassemblers are being extended to support various disassembly tasks on all kinds of binaries. This indicates that, besides recall and precision, it is also important to have ground truth approaches that are general and extendable. Regarding generality, tracing the compiling process is much better than using the intermediate compiler outputs, considering that it can capture anything the compiler knows about without restrictions from the existing implementation of the compiler. Extendibility wise, the two approaches are similarly restricted. To extend support

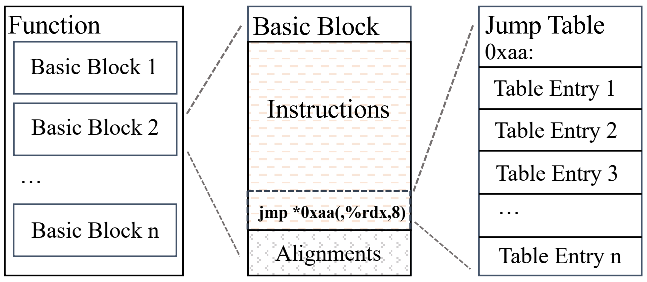


Figure 4: Metadata used by ORACLEGT to track the ground truth of instructions, functions, and jump tables. `Function` represents a function, which consists of a set of `Basic Blocks`. Each basic block has two regions, the instructions and alignment bytes (if any). A basic block also has a field indicating whether it can fall through. An indirect jump that encodes a `Jump Table` is also marked with the list of targets.

for other compilers or architectures, both approaches need to extend the compilation toolchain.

Conclusion. Overall, tracing the compiling process is the approach that can best satisfy today's need. However, the only tool [26] supporting this approach still has two major limitations. First, the tool is not providing results with perfect recall, despite it shall never make mistakes. It cannot cover the ground truth that the compiler cannot recognize, including the aforementioned code encoded as data, functions without a proper label, and handwritten jump tables. It can also miss ground truth that needs more fine-grained tracing, such as the overlapped instructions. Second, the tool currently can only support x64/x86 binaries. It urgently needs an extension to support other architectures, in particular those gaining more popularity nowadays (e.g., ARM and MIPS).

6 Towards Better Ground Truth

Inspired by our discussion above, we further aim to extend the tool presented in [26] to provide better ground truth for binary disassembly. For simplicity of presentation, we will call the tool ORACLEGT. In the rest of this section, we first introduce the technical background of ORACLEGT and then explain our improvements and extensions. Finally, we present an application of the extended ORACLEGT to show its benefits.

6.1 Background

ORACLEGT supports both GCC and Clang for x86/x64 binaries. Considering the internal mechanisms for both GCC and Clang are highly similar, we hereby use GCC as the example to explain how ORACLEGT works. At the high level idea, ORACLEGT instruments the GCC front-end, the GAS assembler, and the gold linker to insert metadata structured as Figure 4 to the outputs at different compilation stages.

```
1 .bbInfo_FUNB
2 .bbInfo_BB
3 pushq   %r15
4 subq    $8, %rsp
5 xorl    %edi, %edi
6 ...
7 call    as_fatal
8 .bbInfo_BE 0
9 .bbInfo_FUNE
```

```
1 cmpl    $10, %r8d
2 ja      .L1021
3 mov    rax, .L1001(,%r8,8)
4 jmp    *rax
5
6 .bbInfo_JMPTBL 11 8
7 .L1001
8 .quad .L1003
9 ...
```

Listing 8: An example of assembly code with directives inserted by ORACLEGT. In the *left* part (i) `.bbInfo_FUNB` and `.bbInfo_FUNE` label the range of a function; (ii) `.bbInfo_BB` and `.bbInfo_BE` label the range of a basic block; (iii) `0` after `.bbInfo_BE` means the current basic block does not fall through. In the *right* part, `.bbInfo_JMPTBL` indicates that the data below represents a jump table, and the follow-up `11` and `8` mean the jump table as 11 entries with every entry of 8 bytes.

Compiling. When GCC front-end finishes the final RTL pass [2], it will output the assembly code to be processed by the assembler. In this step, GCC controls the locations of functions, basic blocks, and instructions. Intercepting the process, ORACLEGT inserts extra directives to label the three constructs. To better illustrate the idea, we show an example in Listing 8. One thing worth mentioning is that every function in the assembly file is contiguous. The later processing by the assembler may split a function into different regions.

Assembling. When the assembly file is fed into the GAS assembler, ORACLEGT intercepts the assembly process to transfer the inserted directives into another form of metadata.

- *Functions:* When encountering a `.bbinfo_FUNB` directive during assembly, ORACLEGT initializes a `FUNC` structure and records the offset from the function start to the current *fragment* (the basic unit that assemblers use to store contiguous code or data). When a follow-up `.bbInfo_FUNE` is met, ORACLEGT marks the end of the current function in the `FUNC` structure.
- *Basic Blocks:* When a `.bbInfo_BB` directive is reached between a `.bbinfo_FUNB` and a `.bbInfo_FUNE`, ORACLEGT creates a `BBL` structure to record the offset from the current basic block to the current fragment. When the corresponding `.bbInfo_BE` appears, ORACLEGT will record whether the basic block falls through in the `BBL` structure. If the basic block contains an alignment region, ORACLEGT will further record the location and size of the alignment. The `BBL` structure, when completed, will be appended to the current function.
- *Instructions:* When an instruction is assembled, ORACLEGT gets its size and updates the size information in the `BBL` structure of the current basic block.
- *Jump Tables:* When a `.bbInfo_JMPTBL` directive is seen, ORACLEGT creates a `JMPTBL` structure to track the location of the jump table (the offset from the jump table to the current fragment), the number of entries, and the size of each entry. Further, ORACLEGT records the reference(s) to the jump table. For instance, in the example shown as Listing 8, ORACLEGT will record the reference to the jump table at line 3 (`.L1001`). Note that no extra efforts are needed to gather the references as ORACLEGT internally collects them from the assembler.

At the end of assembling a file, the assembler will finalize the location of each fragment in the object file. At this point, ORACLEGT updates the `FUNC`, `BBL`, and `JMPTBL` structures to replace the offset to the current fragment with the offset to the object file. All the structures are then organized as an `.gtinfo` section in the object file.

Linking. In this stage, the linker merges different object files to generate the final binary. ORACLEGT instruments this linking process to finalize the addresses of functions, basic blocks, jump tables encoded in the `FUNC`, `BBL`, and `JMPTBL` structures. It also merges the `.gtinfo` section from all object files into a single `.gtinfo` section in the binary. Alignments inserted by the assembler between object files are also recorded into the `.gtinfo` section.

Extracting. Based on the metadata stored in the `.gtinfo` section, ORACLEGT extracts the ground truth information. Specifically, function and basic block information can be easily obtained by reading the `FUNC` and `BBL` structures. Given the range of a basic block, ORACLEGT runs linear disassembly in the code region to collect the instructions. ORACLEGT also records alignment information. This way, when disassembly identifies a instruction inside an alignment region, ORACLEGT can mark it as a "harmless" false positive.

The extraction of jump tables is a bit more complex. Based on the metadata, we only know the location and size of each jump table and the instruction(s) referencing the jump table. There is no direct information about which indirect jump uses the jump table. To pinpoint the indirect jump, ORACLEGT runs forward taint analysis to track the propagation of the reference to the jump table until an indirect jump. Consider Listing 8 as an example. ORACLEGT will track the reference to `.L1001` at line 3 all the way to the indirect jump at line 4. The details of our taint analysis are presented in algorithm 1.

6.2 Improvements

Our major improvement is a post-compilation analysis to identify three sets of missing ground truth.

Instructions Encoded as Data. As demonstrated in Listing 3, the developers can encode instructions as data, which shall mislead the compiler. We extend ORACLEGT to identify those instructions. Basically, we reconstruct the control flows of each function reported by ORACLEGT and collect direct control transfers to regions that are considered data by the compiler. On identifying such a control transfer, we run recursive disassembly from the target location to gather the missing instructions. To ensure correctness, our disassembly is conservative. In particular, we skip all indirect control transfers whose targets remain unknown and we assume a function does not return if we are unsure.

Applying our extension to the x86/x64 benchmark presented in [26], we observed 1,378 data regions embedded in code and 1,356 of them are actually instructions. We manually examined the remaining data regions and believe none of them carries instructions.

Overlapped Instructions and Missed Functions. Native ORACLEGT can not mark overlapped instructions (Listing 4) and functions without a proper label (Listing 5). We use the same approach to handle both cases. Checking the targets direct jumps and jump tables reported by ORACLEGT, we verify whether the targets point to the middle of other instructions. If so, we consider them overlapped instructions. Similarly, we identify direct calls whose targets do not point to a known function and consider those targets previously missed function starts. Running the extension on the above benchmark, ORACLEGT discovers 2,108 more overlapped instructions and 76 missed functions.

Handwritten Jump Tables. Developers may also manually create jump tables in handwritten assembly files. No standard labels are available to notate such jump tables and thus, the compiler cannot recognize them. We propose a new method to identify handwritten jump tables based on three insights. First, the targets of such a jump table should have data-to-code (d2c) references that are contiguously arranged in the data region. Second, the assembly code should contain a code-to-data (c2d) reference to the base address the jump table. Third, the base address should propagate to an indirect jump. Below elaborates on the specifics of our method.

- *Step-1:* We enumerate all the available d2c references to pinpoint the contiguous ones pointing to the same function in handwritten assembly. All the d2c references are natively gathered by ORACLEGT from the compiler, which requires no extra operations. We consider those d2c references potential jump table targets.

- *Step-2:* We visit the c2d references and identify those pointing to a d2c reference collected in Step-1. Each of the c2d references is considered the base address of a jump table.

- *Step-3:* We run forward taint analysis from each c2d reference determined at Step-2 and validate whether it propagates to the target of an indirect jump. If that happens, we consider the c2d reference points to a real jump table enforced at the indirect jump.

Using the above method on the benchmarks presented in [26] again, we collected 1,882 jump tables missed by ORACLEGT.

6.3 Extending

Besides improving the recall of ORACLEGT, we further extended its support on both GCC/Clang for other popular architectures, including ARM32, AArch64, MIPS32 and MIPS64. To realize the extension, we do not need to change the compiler front-end and the linker since ORACLEGT's operations at the two stages are architecture-independent. The major work is to adapt the modules to assemble ARM32 / AArch64 / MIPS32 / MIPS64 code in the GCC GAS assembler and the Clang integrated assembler. The adaptation is mostly straightforward by following the procedure described in subsection 6.1. Below we discuss some details worth mentioning.

Adding Type Information of Basic Block. ARM32 includes two execution modes, ARM mode and Thumb mode [7]. The encoding of instructions are different under the two modes. Thereby the detection of execution mode is a critical task for ARM disassemblers. To support evaluation of this task, we extend ORACLEGT to add an extra directive to mark the execution mode of ARM32 basic blocks. Knowledge of the execution mode can be easily obtained by querying the GCC GAS assembler or the Clang integrated assembler.

```
1 cmp    $0x57,%rdx
2 ja     .Ldefault
3 jmpq   *.LJMPTBL(,%rdx,8)
4 ...
5 .LJMPTBL:
6 entry 1
7 ...
8 entry n1
```

```
1 cmp    r5, #3
2 ldrls  pc, [pc,r5,lsl #2]
3 b      .Ldefault
4 .LJMPTBL
5 entry 1
6 entry 2
7 ...
8 entry n2
```

Listing 9: Examples of explicit reference and implicit reference to jump tables. The *left* shows a x64 example which contains an explicit reference to the jump table (line 3). The *right* part is an example of ARM32, there is no explicit reference to the jump table. Instead, the reference to jump table is held by `pc` in `[pc, r5, lsl #2]`(line 2) implicitly.

Detecting Implicit References to Jump Tables. As pointed out in subsection 6.1, ORACLEGT relies on explicit references to a jump table to locate the corresponding indirect jump. However, a jump table may not always be explicitly referenced in ARM binaries (i.e., the references may not be recognized by the compiler). The *right* part of Listing 9 shows one such example. To handle such cases, we design a method as follows. Given a jump table without explicit references, we visit the use of the `pc` register in the assembly code placed before the jump table and inspect whether the `pc` register is an implicit reference to jump table. If so, we create a dummy reference to mark the implicit reference.

Taint Analysis Across Memory. When extracting the ground truth of jump tables, ORACLEGT runs taint analysis to track the propagation of the reference(s) to the jump table. On x86/x64 binaries, it is sufficient to only track the registers.

However, on ARM/MIPS binaries, the referenced value can propagate through the stack. Thus, we extend the taint analysis in ORACLEGT by adding the supports of taint propagation among stack memories.

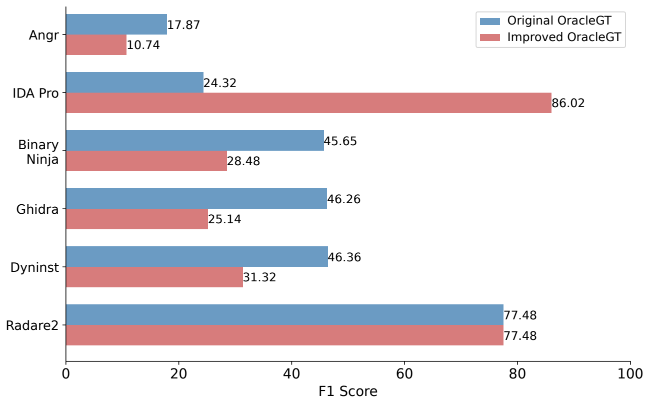
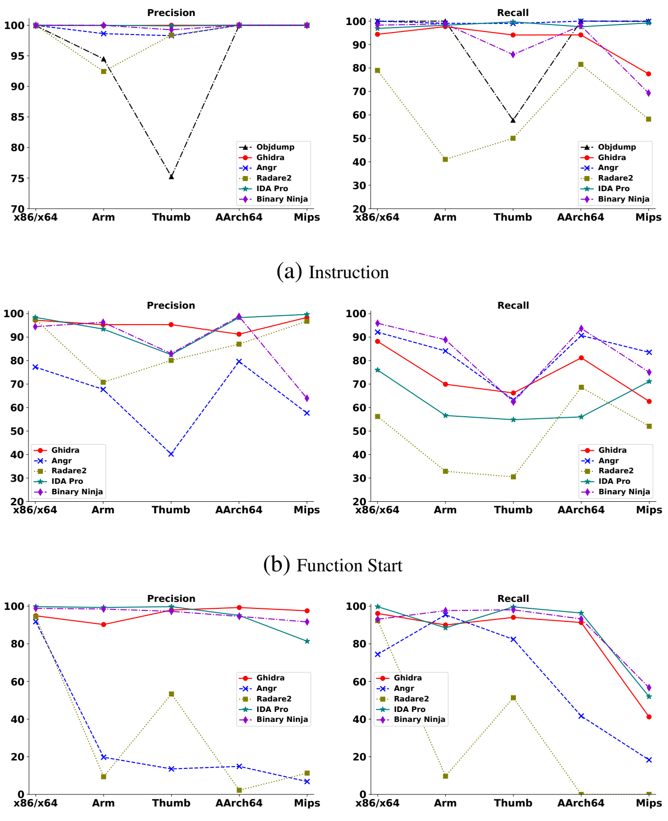


Figure 5: Accuracy (F-1 score) of popular disassemblers on recovering jump tables from `glibc`, measured using the original ORACLEGT and our improved ORACLEGT.

Discussion. To illustrate the benefits of our improvement to ORACLEGT, we perform a small case study. Specifically, we measure the accuracy of six popular disassemblers on recovering jump tables from `glibc`, separately using the original ORACLEGT and our improved ORACLEGT for ground truth. We focus on jump tables and `glibc` as the impact of our improvement in this setup is more evident. Figure 5 shows the evaluation results. The key observation is that our improvement to ORACLEGT meaningfully helps understand the true accuracy of existing disassembly tools.

While our improvement is beneficial, it is still not perfect. All our improvement strategies are designed to be conservative, which are error-free and maintain the correctness of ORACLEGT. However, the strategies, when coupled with ORACLEGT, can still miss cases, which we discuss as follows.

- *Instructions:* Given functions that (i) cannot be recognized by the compiler and (ii) are not directly called by any other recovered functions, we cannot realize their existence. Thus, we will miss their instructions. In addition, we will miss instructions that can only be reached by the fall-through edge of a function call whose return status is unknown. This is because we conservatively assume a function does not return unless we know for sure.
- *Functions:* As described above, we cannot identify functions that (i) cannot be recognized by the compiler and (ii) are not directly called by any other recovered functions.
- *Jump Tables:* We perform taint analysis to pinpoint the indirect jump pertaining to each jump table. Inspired by empirical observations, we only track taints propagated across stack memory and registers. In theory, we can under-taint a target that propagates through non-stack memory and thus, miss the corresponding jump table.



(a) Instruction

(b) Function Start

(c) Jump Table

Figure 6: Recall and precision of mainstream disassemblers on binaries with different architectures.

6.4 Application

Leveraging our extensions, we rebuild the testsuites presented in [26] with both GCC-8.1 and Clang/LLVM-6.0 to run on ARM32 (with ARM and Thumb mode), AArch64, MIPS32 and MIPS64 architectures. For every program, we build it with various optimization levels (O2, O3, Os, Ofast), which finally generates a testsuite with 6,328 binaries. Based on the testsuite, we re-evaluate OBJDUMP-2.30 [14], GHIDRA-9.04 [24], ANGR-8.19.5.25 [9], RADARE2-4.4.0 [30], IDA PRO-7.4 and BINARY NINJA-1.2 with all binaries stripped. Figure 6 shows the average results of instruction recover, function start detection, and jump table reconstruction by different disassembly tools. The actual numbers are presented in Table 7 in the Appendix. According to the evaluation, we summarize three observations that were previously less-known.

The performance of modern disassemblers vary across architectures. As shown in Figure 6, modern disassemblers present very inconsistent performance when applied to binaries running on different architectures. Overall, they all present high recall and precision on x86/x64 targets, regardless of the disassembly tasks. We envision two major reasons leading to this phenomenon. On the one hand, disassemblers are mostly created to work on x86/x64 binaries. They have received longer and broader improvement to handle x86/x64 binaries. On the other hand, benchmarks and ground truth approaches to evaluate disassembly are better ready for x86/x64 binaries [5, 26], greatly benefit the evolving cycle.

ARM32 poses a bigger challenge than AArch64 to modern disassemblers. Many disassemblers show limitations when handling ARM32 binaries, in particular the code running in the Thumb mode. This has been similarly unveiled by a recent study *et al.* [17]. However, no research has been conducted to run a comparison between ARM32 and AArch64 like what we show in Figure 6. In most of the cases, disassemblers perform significantly better on AArch64 binaries than on ARM32 binaries. While this is not surprising considering that ARM32 uses the mixed modes of ARM and Thumb, our evaluation brings quantitative evidence to back the observation.

```
410980 <quotearg_buffer>:
...
; initialize gp
; at = 0x2 << 16
lui     at,0x2
daddu   at,at,t9
daddiu  gp,at,30512
...
```

```
; load base of jmptbl
ld      v1,-32720(gp)
...
daddu   at,at,v1
; load entry of jmptbl
ld      at,-16952(at)
daddu   at,at,gp
jr      at
```

Listing 10: Examples of jump table in `coreutils-8.30` compiled by Clang-6.0 for MIPS. The *right* part is an example which calculates the target of indirect jumps depending on the value of gp. The *left* part shows that gp is initialized at the beginning of the function. Its calculation is explained as follows: When calling a function, `t9` stores the address of the called function. Thus at the beginning of a function, `t9` holds the address of current function. In this example, the value of `t9` is `0x410980`. In order to calculate the value of gp, disassembler should recover the correct address of function.

Commercial disassemblers are less effective with handling MIPS binaries. Commercial disassemblers, like IDA PRO and BINARY NINJA, are widely believed to be very powerful in dealing with modern binaries. This is very true on x86/x64 binaries. However, opposite observations arise when we switch them to handle MIPS binaries. Both IDA PRO and BINARY NINJA present massively reduced utilities when disassembling MIPS binaries. In particular, when recovering jump tables, the precision and recall of IDA PRO drop to 81.39% and 51.98%. BINARY NINJA has a higher precision (91.52%) but its recall is also only 56.56%.

Motivated to understand why the commercial tools have a downgrade in performance, we manually inspected a set of jump tables missed by both IDA PRO and BINARY NINJA. We found that the calculation of jump table targets relies on the value of register gp[4] as shown in Listing 10. And the value of gp is calculated at the beginning of a function based on the address of the current function. However, both IDA PRO and BINARY NINJA miss plenty of function starts on MIPS binaries, indirectly causing the low recall of jump tables.

7 Related works

Generating Ground Truth for Binary Disassembly. As an essential step toward many binary analysis techniques, collecting ground truth for binary disassembly has attracted increasing attention. One line of researches [5, 6, 10, 17, 28, 29, 35] follow a so-called standard strategy which uses address embedded in symbols as the ground truth. However, as discussed in §3, ground truth relying on symbols is nether accurate nor complete. In comparison, Meng *et al.* [22] manually collected ground truth for a small set of instructions, functions, and control flows. This approach can achieve extremely high precision but is less popular due to the scalability limitation. Meanwhile, another line of researches [18, 23] rely on ground truth generated by the existing disassemblers. For instance, Nagy *et al.* [23] leverage disassembling result from OBJDUMP as the baseline while Kinder *et al.* [18] directly reuse the jump tables detected by IDA Pro. Unfortunately, as shown in Pang *et al.* [26], none of the existing disassembler does perfect on various binary challenges, which leads to the fact completely trust disassemblers has more disadvantages than benefits.

On the other hand, more compelling and reliable approaches generating ground truth relies on either the intermediate compiler outputs or compilation metadata. For instance, David *et al.* [34] dump the intermediate representation of GCC and extract the rough data of jump tables, but they could not map the information to the final executable files. [5, 28] generate ground truth by performing conservatively linear sweeping between continuous regions based on debug line information. Failing to take unreachable alignment code into consideration, Andriesse *et al.* 's approach [5] still misses about 2% instructions. In contrast, Li *et al.* [21] leverage the listing files produced by GNU Assembler to extract the ground truth, but they struggle on mapping the instructions into final executable files. As the most closely related work to ours, Pang *et al.* [26] collect the ground truth by tracing compilation process, which guarantees high precision and recall. As a work in a later position, we not only extend the tracing compilation process approach to more architectures but also fix corner cases missed by Pang *et al.* [26].

Discussion on Ground Truth for Binary Disassembly. A

[4]gp is used as a global pointer pointing to the midst region of 64K static data [1], which is initialized at runtime.

recent study by Alves-Foss *et al.* [4] discusses challenges in defining and identifying ground truth for binary disassembly. The study further lists some common approaches to generating the ground truth and showcases issues related to those approaches. Our paper can be viewed as the next step of this study. We present a systematic taxonomy and comparison of the existing approaches, unveil the implications behind the imperfection of those approaches, pinpoint the more appropriate approach, and shed light on building trustworthy ground truth for binary disassembly.

Evaluation of Binary Disassembly. Binary disassembly is a critical task for binary analysis. Recently, many researches [5,10,17,21,26,32] have made great effort on binary disassembly evaluations. Jinag *et al.* [17] built testsuite on Arm32 and evaluate the performance of disassemblers on instructions recovery. Li *et al.* [21] evaluate the performance of instructions recovery among different disassemblers on x86/x64. [5, 26] build large scale testsuite on x86/x64 to evaluate the performance of disassemblers on instructions, function start and jump tables recovery. [6, 10, 32] evaluate function starts identifications on x86/x64. All above mentioned papers mainly focus on evaluating binary disassembly within a specific architecture. As a compensation, our paper concentrates on improving the ground truth used by binary disassembly on all popular architectures.

8 Conclusion

This paper concerns the approaches to generating ground truth for binary disassembly. We bring a taxonomy of the approaches used by past research, unveiling the mechanisms behind the approaches. Throughout a systematic comparison of the mechanisms, we present a deep understanding of the mechanisms regarding the key properties (recall, precision, generality, and extendibility) that affect their applications. In a follow-up empirical evaluation, we further validate that defects in those properties can significantly hurt the applications and even lead to misleading conclusions. Finally, we identify and rectify the limitations of the tool that carries the best potential to meet those properties. This effort is expected to benefit various use scenarios of disassembly ground truth. In particular, we demonstrate that using this tool to re-evaluate the mainstream disassemblers leads to many previously less-known observations. We hope this piece of research can provide references and tools to standardize and unify the ground truth in binary disassembly evaluation.

Acknowledgements

We thank our shepherd Tiffany Bao and anonymous reviewers for their feedback. We also thank Kexin Pei for helping with the experiment on XDA. Chengbin Pang, Tiantai Zhang, and Bing Mao were supported by Chinese National Natural Science Foundation(Grant#: 61272078, 62032010, 62172201). Ruotong Yu and Jun Xu were partially supported by the Office of Naval Research (Grant#: N00014-17-1-2787). Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agency.

References

[1] Mips assemble language. https://training.mips.com/basic_mips/PDF/Assemble_Language.pdf, 2021.

[2] Rtl passes, gnu compiler collection (gcc) internals. https://gcc.gnu.org/onlinedocs/gccint/RTL-passes.html, 2022.

[3] National Security Agency. Ghidra. https://www.nsa.gov/resources/everyone/ghidra/, 2021.

[4] Jim Alves-Foss and Varsha Venugopal. The inconvenient truths of ground truth for binary analysis. In *Workshop on Binary Analysis Research (BAR)*, 2022.

[5] Dennis Andriesse, Xi Chen, Victor Van Der Veen, Asia Slowinska, and Herbert Bos. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 583–600, 2016.

[6] Dennis Andriesse, Asia Slowinska, and Herbert Bos. Compiler-agnostic function detection in binaries. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 177–189. IEEE, 2017.

[7] ARM. Arm architecture reference manual armv7-a. https://developer.arm.com/documentation/ddi0406/latest, 2021.

[8] ARM. Arm mapping symbols. https://developer.arm.com/documentation/dui0803/a/Accessing-and-managing-symbols-with-armlink/About-mapping-symbols, 2021.

[9] Computer Security Lab at UC Santa Barbara. angr github repo. https://github.com/angr/angr/tree/76da434f, 2020.

[10] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. {BYTEWEIGHT}: Learning to recognize functions in binary code. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 845–860, 2014.

[11] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan.

Bingo: Cross-architecture cross-os binary search. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 678–689, 2016.

[12] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z Snow, and Fabian Monrose. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In *NDSS*, 2015.

[13] Chris Eagle. *The IDA pro book*. No Starch Press, 2011.

[14] GNU. Objdump 2.30. https://ftp.gnu.org/gnu/binutils/binutils-2.30.tar.xz, 2020.

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

[16] Yikun Hu, Yuanyuan Zhang, Juanru Li, and Dawu Gu. Binary code clone detection across architectures and compiling configurations. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 88–98. IEEE, 2017.

[17] Muhui Jiang, Yajin Zhou, Xiapu Luo, Ruoyu Wang, Yang Liu, and Kui Ren. An empirical study on arm disassembly tools. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 401–414, 2020.

[18] Johannes Kinder and Dmitry Kravchenko. Alternating control flow reconstruction. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 267–282. Springer, 2012.

[19] Hyungjoon Koo, Yaohui Chen, Long Lu, Vasileios P Kemerlis, and Michalis Polychronakis. Compiler-assisted code randomization. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 461–477. IEEE, 2018.

[20] Aditya Kumar. Hot cold splitting in llvm. https://llvm.org/devmtg/2019-10/slides/Kumar-HotColdSplitting.pdf, 2019.

[21] Kaiyuan Li, Maverick Woo, and Limin Jia. 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*, pages 9–14, 2020.

[22] Xiaozhu Meng and Barton P Miller. Binary code is not easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 24–35, 2016.

[23] Stefan Nagy, Anh Nguyen-Tuong, Jason D Hiser, Jack W Davidson, and Matthew Hicks. Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[24] NationalSecurityAgency. Ghidra 9.04. https://htmlpreview.github.io/?https://github.com/NationalSecurityAgency/ghidra/blob/Ghidra_9.0.4_build/Ghidra/Configurations/Public_Release/src/global/docs/ChangeHistory.html, 2020.

[25] Binary Ninja. binary.ninja : a reverse engineering platform. https://binary.ninja/, 2021.

[26] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. Sok: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 833–851. IEEE, 2021.

[27] Chengbin Pang, Ruotong Yu, Dongpeng Xu, Eric Koskinen, Georgios Portokalidis, and Jun Xu. Towards optimal use of exception handling information for function detection. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 338–349. IEEE, 2021.

[28] Kexin Pei, Jonas Guan, David Williams-King, Junfeng Yang, and Suman Jana. Xda: Accurate, robust disassembly with transfer learning. *arXiv preprint arXiv:2010.00770*, 2020.

[29] Rui Qiao and R Sekar. Function interface analysis: A principled approach for function recognition in cots binaries. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 201–212. IEEE, 2017.

[30] radreorg. Radare2 github repo. https://github.com/radareorg/radare2/tree/5a1df188, 2020.

[31] Pawel Sarbinowski, Vasileios P Kemerlis, Cristiano Giuffrida, and Elias Athanasopoulos. Vtpin: practical vtable hijacking protection for binaries. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 448–459, 2016.

[32] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. Recognizing functions in binaries with neural networks. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 611–626, 2015.

[33] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher

Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157. IEEE, 2016.

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

[35] Sheng Yu, Yu Qu, Xunchao Hu, and Heng Yin. Deepdi: Learning a relational graph convolutional network model on instructions for fast and accurate disassembly.

[36] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. Order matters: Semantic-aware neural networks for binary code similarity detection. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 1145–1152, 2020.

[37] Mingwei Zhang and R Sekar. Control flow integrity for {COTS} binaries. In *22nd {USENIX} Security Symposium ({USENIX} Security 13)*, pages 337–352, 2013.

A ORACLEGT *v.s.* Compilation Metadata

In this section, we present a comparison between ORACLEGT and the approach present in [5] (which leverages debug information to extract ground truth for binary disassembly). Specifically, we run ORACLEGT and [5] in their conservative mode to collect the ground truth of instructions in the x86/x64 benchmarks presented in [26], and we compare the results of both approaches. As ORACLEGT is ensured to be correct, we consider every instruction identified by ORACLEGT but missed by [5] as a false negative (FN) of [5]. We further manually examined the instructions detected by [5] but not covered by ORACLEGT. We confirmed all of them are false positives (FPs) of [5]. As we can see in Table 6, [5] can incur tremendous FNs and a meaningful group of FPs, presenting a lower utility than ORACLEGT. As shown in Listing 6, debug information only carries the locations of some instructions but not of them and [5] only runs conservative disassembly between two continuous locations, which leads to the plenty of false negatives. Further, as illustrated in Listing 7, the debug information can be inaccurate and thus, also leads to false positives of [5].

Algorithm 1: FIND INDIRECT JUMPS.

```
Input  : A list of cross reference to identified jump table:
         $\vec{JTR} = \{jtr_1, jtr_2, ..., jtr_n\}$
Input  : Control flow graph of functions: $CFG$
Output : A list of mappings between cross references to jump
         table and indirect jumps:
         $\vec{M} = \{(jtr_1, ij_1), (jtr_2, ij_2)..., (jtr_n, ij_n)\}$
/* $ij_i$ represents ith founded indirect jump. */
Procedure taint_instruction($I$):
    tainted = false
    for each register $\mathcal{R}_r$ used for reading in $I$ do
        if $\mathcal{R}_r$.is_tainted() then
            tainted = true
        end
    end
    for each register $\mathcal{R}_w$ used for writing in $I$ do
        if tainted then
            $\mathcal{R}_w$.taint()
        else
            $\mathcal{R}_w$.clear_taint()
        end
    end
return
Initialization: $\vec{M} = \emptyset$; $fixpoint = false$
while $\neg fixpoint$ do
    /* Loop until $\vec{CFG}$ could not not updated. */
    $fixpoint = true$
    for each $jtr_i$ in $\vec{JTR}$ do
        $\vec{Q} = \emptyset$
        $I = CFG.get\_instr(jtr_i)$ /* get the instruction
            contains $jtr_i$ */
        $I.taint\_initialize()$
        $\vec{Q}.push(I)$
        while $\vec{Q}$.is_not_empty() do
            $I = \vec{Q}.pop()$
            $taint\_instruction(I)$
            if $I$.is_tainted() and $I$.is_indirect_jump() then
                $fixpoint = false$
                $\vec{M}.add((jtr_i, instruction))$
                $\vec{CFG}$.update($I$, $jtr_i$) /* update $\vec{CFG}$
                    according to the jump table
                    information */
                $\vec{JTR}$.remove($jtr_i$)
                break
            end
            $\vec{Q}$.append($CFG$.get_successors($I$))
        end
    end
end
return $\vec{M}$;
```

Table 6: The number of FPs and FNs incurred by [5].

OPT	O0	O1	O2	O3	Os	Of
# of FPs	301	361	501	690	361	908
# of FNs	2,764K	898K	444K	471K	497K	466K

Table 7: Evaluation results of instruction recovery, function start detection, and jump table reconstruction by mainstream disassemblers on ARM32/AArch64/MIPS binaries. In the colloumns, **O** indicates the optimization level; **T** means ARM32 Thumb mode; `Pre` and `Rec` represent precision and recall. We merge the results of MIPS32 and MIPS64 as their instruction encoding is similar. The best/worst results specific to each optimization level are respectively marked in blue/red color.

	Arch		**O**	**Objdump**		**Ghidra**		**Angr**		**Radare2**		**IDA**		**Ninja**	
				`Pre`	`Rec`	`Pre`	`Rec`	`Pre`	`Rec`	`Pre`	`Rec`	`Pre`	`Rec`	`Pre`	`Rec`
Instructions	**ARM**	**32**	O2	93.77	99.99	99.95	97.57	99.04	99.53	92.79	41.24	99.99	97.97	99.99	99.03
			O3	94.42	99.99	99.96	97.79	98.93	99.46	91.41	40.72	99.99	98.09	99.99	98.85
			Os	95.95	99.99	99.97	97.54	97.19	98.42	91.93	42.80	99.99	98.89	99.98	98.50
			Of	94.47	99.99	99.94	97.69	99.01	99.47	92.26	39.32	99.99	98.05	99.99	98.87
		T	O2	72.25	57.56	99.93	93.31	98.82	99.03	98.34	46.29	99.90	97.29	99.65	86.57
			O3	73.10	57.27	99.94	89.24	98.78	99.02	98.91	45.07	99.90	97.33	99.64	83.56
			Os	75.26	57.39	99.91	92.93	96.80	98.94	99.24	48.45	99.89	97.77	99.70	90.13
			Of	73.14	57.28	99.94	89.16	98.78	99.02	98.79	46.47	99.90	97.32	99.67	83.59
		64	O2	100.0	100.0	100.0	94.21	100.0	99.99	100.0	79.07	100.0	97.48	100.0	98.29
			O3	100.0	100.0	100.0	93.70	100.0	100.0	100.0	83.47	100.0	97.72	100.0	98.20
			Os	100.0	100.0	100.0	94.30	100.0	100.0	100.0	84.02	100.0	97.68	100.0	97.94
			Of	100.0	100.0	100.0	94.10	100.0	100.0	100.0	79.64	100.0	97.54	100.0	97.94
	Mips	**32 & 64**	O2	100.0	100.0	100.0	77.88	100.0	99.85	100.0	57.15	100.0	99.10	99.99	68.04
			O3	100.0	100.0	100.0	74.11	100.0	99.85	100.0	54.46	100.0	99.36	99.99	69.24
			Os	100.0	100.0	100.0	83.52	100.0	99.86	100.0	67.16	100.0	99.06	99.99	71.20
			Of	100.0	100.0	100.0	74.44	100.0	99.85	100.0	54.58	100.0	99.37	99.99	67.85
Functions	**ARM**	**32**	O2	–	–	95.12	68.42	71.39	85.50	69.43	32.83	93.60	56.75	96.31	88.98
			O3	–	–	94.93	70.97	68.14	83.75	69.28	32.67	93.29	54.75	96.24	88.91
			Os	–	–	96.12	69.40	63.43	83.90	72.49	34.11	93.48	60.52	96.43	87.96
			Of	–	–	95.04	70.91	68.27	83.30	71.94	32.69	93.34	54.51	96.29	88.76
		T	O2	–	–	95.42	67.25	42.63	62.64	79.77	30.71	83.67	55.60	84.07	62.13
			O3	–	–	95.39	66.55	39.47	61.47	79.60	29.83	83.55	53.48	79.87	60.45
			Os	–	–	95.05	65.00	42.17	70.33	81.01	32.51	84.22	59.15	87.64	66.19
			Of	–	–	95.47	66.54	39.49	61.22	79.91	30.35	83.68	53.26	80.49	60.74
		64	O2	–	–	89.55	80.75	78.14	89.23	87.97	67.94	99.35	56.61	99.08	94.73
			O3	–	–	88.75	80.88	75.92	87.83	86.36	68.88	98.73	54.33	98.81	93.58
			Os	–	–	97.74	81.96	89.08	97.83	87.14	69.16	95.02	57.49	97.82	92.64
			Of	–	–	88.91	81.17	75.61	88.02	86.40	69.41	99.36	55.51	98.81	93.73
	Mips	**32 & 64**	O2	–	–	98.60	62.33	56.42	83.32	96.05	51.99	99.81	71.27	64.27	75.26
			O3	–	–	98.22	58.76	54.72	81.13	96.06	49.35	99.81	71.28	63.20	73.82
			Os	–	–	98.81	69.99	65.76	89.00	99.58	58.17	99.86	76.25	65.89	78.24
			Of	–	–	98.21	59.25	53.83	81.23	96.10	49.45	99.82	71.62	62.91	73.36
Jump Tables	**ARM**	**32**	O2	–	–	93.87	93.77	22.12	95.85	10.27	10.48	99.29	95.19	98.65	97.97
			O3	–	–	87.04	86.62	21.14	94.47	11.17	11.75	98.99	81.54	98.58	97.14
			Os	–	–	93.27	93.30	14.31	96.02	6.33	5.94	99.59	95.68	98.47	98.24
			Of	–	–	86.85	86.43	21.31	95.14	9.82	10.49	99.04	81.67	98.67	97.25
		T	O2	–	–	98.57	97.54	15.41	83.42	50.16	46.83	99.63	99.50	99.19	98.62
			O3	–	–	98.59	91.52	14.69	82.58	59.95	55.77	99.73	99.60	98.21	97.45
			Os	–	–	97.63	97.29	9.51	81.11	47.63	46.57	99.91	99.75	98.98	98.84
			Of	–	–	98.50	91.14	14.62	82.44	59.86	56.34	99.73	99.60	98.21	97.45
		64	O2	–	–	99.32	86.62	14.47	41.06	11.11	0.01	96.16	98.17	98.63	95.74
			O3	–	–	99.23	95.03	15.28	41.34	0.00	0.00	94.88	96.01	98.55	98.11
			Os	–	–	99.45	98.36	15.05	41.97	0.00	0.00	93.12	94.05	83.49	83.41
			Of	–	–	99.17	85.31	14.41	40.96	0.00	0.00	95.55	96.71	97.65	95.69
	Mips	**32 & 64**	O2	–	–	97.74	41.90	6.94	19.06	14.28	0.01	82.09	51.38	94.59	57.32
			O3	–	–	97.30	36.64	7.27	16.19	14.28	0.01	81.59	51.81	91.14	55.73
			Os	–	–	97.58	49.04	5.31	19.97	16.66	0.01	80.32	52.92	92.45	58.51
			Of	–	–	97.31	37.51	7.82	16.45	0.00	0.00	81.58	51.82	87.90	54.70