On the Generation of Disassembly Ground Truth and the Evaluation of Disassemblers

Kaiyuan Li
squid@cmu.edu
Carnegie Mellon University, CyLab

Maverick Woo
pooh@cmu.edu
Carnegie Mellon University, CyLab

Limin Jia
liminjia@cmu.edu
Carnegie Mellon University, CyLab

ABSTRACT

When a software transformation or software security task needs to analyze a given program binary, the first step is often disassembly. Since many modern disassemblers have become highly accurate on many binaries, we believe reliable disassembler benchmarking requires standardizing the set of binaries used and the disassembly ground truth about these binaries. This paper presents (i) a first version of our work-in-progress disassembly benchmark suite, which comprises 879 binaries from diverse projects compiled with multiple compilers and optimization settings, and (ii) a novel disassembly ground truth generator leveraging the notion of "listing files", which has broad support by clang, gcc, icc, and msvc. In additional, it presents our evaluation of four prominent open-source disassemblers using this benchmark suite and a custom evaluation system. Our entire system and all generated data are maintained openly on GitHub to encourage community adoption.

CCS CONCEPTS

• **Security and privacy → Software reverse engineering**; • **Software and its engineering → Assembly languages**.

KEYWORDS

disassembly; ground-truth generation; benchmark suite

ACM Reference Format:

Kaiyuan Li, Maverick Woo, and Limin Jia. 2020. On the Generation of Disassembly Ground Truth and the Evaluation of Disassemblers. In *2020 Workshop on Forming an Ecosystem Around Software Transformation (FEAST '20), November 13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3411502.3418429

1 INTRODUCTION

Many scenarios in software transformation and software security require us to analyze or operate on a given program binary. The most common example is when we do not have access to the source code of the binary. But even when we do, we may not have access to the toolchain or the environment needed to compile the transformed source, or we may be interested in analyses that depend on the actual machine code such as the amount of padding around functions or buffers. In these applications, the first step of the analysis is usually to *disassemble* the binary, which refers to the process of translating its machine code into assembly code.

The study of *disassembly* and the corresponding tool *disassemblers* dates back to at least 1980 [9]. In recent years, the accuracy of many disassemblers is approaching or even exceeding 99% on many real-world binaries—see, e.g., [1] and [8]. Unfortunately, due to well-known problems in binary analysis such as indirect jump resolution and function start identification, the proverbial "last 1%" remains a challenge in disassembly. In response, our community has continued to invent innovative disassembly algorithms, with new work and improvements appearing frequently—see, e.g., [8], [13], and new releases of various existing disassemblers.

Comparing highly-accurate disassemblers is difficult for two reasons. First, since different disassemblers may have different weaknesses, a small change to the set of binaries used to evaluate disassemblers may change their accuracy ranking significantly. Unfortunately, the disassembly literature has so far used different target sets, e.g., [1] vs. [8]. Second, since current disassemblers are exceeding 99% accuracy on many binaries, we need essentially 100%-accurate ground truth to rank the disassemblers reliably. However, as this paper will present, while previous research tended not to discuss their ground truth generation methods at any length, our own experience suggests that this task, if required to be perfectly accurate, has much complexity and is thus not easy to reproduce.

Due to the above reasons, we therefore believe it is high time for our community to start standardizing on a set of community-accepted binaries for benchmarking disassemblers on their accuracy and, out of practicality, their running time and memory usage. To help jumpstart this process, this paper presents a first version of our work-in-progress disassembly benchmark suite. We will discuss our suite from two aspects: (a) the set of binaries included, and (b) our method to generate accurate disassembly ground truth on the instructions in these binaries.

For aspect (a), we believe a good disassembly benchmark suite should have these properties: (i) The included binaries should be *diverse* in size, type (e.g., editor vs. web server), compilers used, and optimization settings used. Achieving these would allow the binaries to better capture the complexities in real-world binaries. (ii) The number of included binaries should be *moderate*, i.e., large enough but not too large. This would ensure the practicality of evaluating disassemblers over every included binary, whose number grows multiplicatively in the number of compilers and optimization settings in each supported ISA-OS pair. At present, our benchmark suite is organized by a notion of "project names", which is an open-source project and a specific version (e.g., `openssh-7.1p2`). For each project name, we specify a specific target binary inside (e.g., `sshd`) and the set of supported ISA-OS pair(s) (e.g., {x86-Linux, x64-Linux}). In addition, for each ISA-OS pair, we specify a list of compiler-version pairs and a list of optimization settings, which will be detailed in §4.2. To encourage adoption, we maintain our benchmark suite openly on GitHub (https://github.com/pangine/disasm-benchmark), with the hope that it will evolve over time in ways similar to the SPEC CPU benchmark [6] due to community inputs and future investigations.

For aspect (b), while one may believe the generation of disassembly ground truth is simply a matter of having the compiler save the assembly code generated (e.g., `gcc -S`), we find this to be an over-simplification in practice. First, even assuming we have all the generated assembly files, in truth the assembly statements (instructions/directives) in these files do *not* form a complete description of the machine code in the binary because some statements admit multiple machine code encodings. Second, when developing our current benchmark suite, we have discovered many corner cases to handle when collecting the generated assembly files, extracting information from them, and representing such information. In §3, we will discuss some of these challenges and our solutions to them.

Finally, using our work-in-progress benchmark suite and our (already mature) ground truth generator and disassembler evaluator, we will present in §4 our findings on four prominent open-source disassemblers: BAP [2], Ghidra [15], Radare2 [18], and ROSE [19]. Our findings largely agree with [1]: (i) function start identification remains a major issue that inhibits accurate disassembly, and (ii) linear-sweep disassembly [21] can be *incidentally* highly accurate, even though it also has no hope of guaranteeing perfect accuracy and thus cannot be relied upon as a long-term solution.

In summary, our contributions in this work are: (1) We propose to start standardizing on a set of binaries from diverse projects compiled with multiple compilers and optimization settings for disassembler evaluation and present our work-in-progress suite as a starting point for community discussion. (2) We developed a new system based on a broadly-supported compiler toolchain feature known as "listing files" to generate accurate disassembly ground truth. We also developed support programs to evaluate four prominent open-source disassemblers against our ground truth and we present our findings. (3) We open-source all our code and data to enable future studies and to encourage community adoption of and contribution to our system and our benchmark suite.

2 BACKGROUND AND RELATED WORK

Disassembly. The problem of disassembly has a long history that dates back to at least 1980 [9]. Nowadays, the most common disassembly algorithms are usually based on either linear sweep or recursive traversal as presented in [21], often with substantial enhancements. For more information of disassembly in general, we refer the reader to more recent works [3, 8, 11, 13] and the references therein. In this paper, we will be evaluating four prominent open-source disassemblers: BAP, Ghidra, Radare2, and ROSE. We note that ROSE comes with multiple disassemblers and we used its recursive traversal implementation. To the best of our knowledge based on reading their source code and official documentation, we believe these disassemblers are all based on recursive traversal.

Prior Work on Disassembly Ground Truth. The topic of disassembly ground truth generation has received surprisingly little space in the literature. With the exception of the work by Andriesse et al. [1], most publications are relatively succinct on their description in this aspect. Here we present a few examples to show three common approaches. (1) Debug info: Miller et al. [13] derived their ground truth from symbol information for ELF ([13, §5]) and PDB for COFF ([13, §5.3]). (2) IDA Pro: Wartell et al. [25, §3] obtained their ground truth using IDA Pro, but they also specifically mentioned they needed manual effort to compare the disassembly results because they noted inaccuracies in the IDA Pro output. (3) Objdump: Khadra et al. [11, §5.2] developed a custom disassembler for ground truth generation and they mentioned validating its result with objdump.

In contrast to the brief descriptions inside the above examples, Andriesse et al. [1, §2.5] dedicated almost an entire column to disassembly ground truth generation. In their paper, they studied 981 real-world x86 and x64 binaries from C/C++ projects compiled using gcc v5.1, clang v3.7, and Visual Studio 2015 with various optimization settings. For each Linux binary, they used a custom LLVM pass to collect source-level information such as source lines belonging to functions and switch statements. Then they used DWARF to link this information to binary offsets and to extract function starts and signatures. Finally, they used a conservative linear-sweep to obtain the ground truth on over 98% of the code bytes. As for the Windows binaries, they briefly mentioned that their ground truth extraction relied on PDB.

Our investigation of their released data and documentation reveals a few limitations. (1) Not extendable by others: Although Andriesse et al. have generously released their build information and ground truth data in full, they did not release their tools and thus their benchmark suite is currently not extendable by others. Our work aims to overcome this. (2) Not fully-automated: Since their ground truth generator did not completely cover the last 2% of the code bytes in their Linux binaries, Andriesse et al. relied on manual analysis to obtain ground truth on those remaining bytes. Since we anticipate our benchmark suite will change over time, we seek a fully automatic approach to increase efficiency and reliability. (3) LLVM requirement: Since their ground truth generator used an LLVM pass to read source-level informations, this restricts the benchmark suite to binaries written in languages that have compiler frontends capable of using LLVM as a backend. Although our current suite contains C projects only, a dependency on LLVM would prohibit future extension to include binaries that do not fit the above criteria (current examples include Go and OCaml).

3 GROUND TRUTH GENERATION

In this section, we present our disassembly ground truth generation method, which is based on the parsing and manipulation of compiler-generated assembly files, object files, listing files, debug information, and the actual binaries. Our description here is Linux-centric even though it covers msvc on Windows. This is made possible because, as part of this project, we have matured the technology of using wine to run `cl` and related Windows tools in Linux to a degree that is sufficient for our Windows tasks.

Scope. The instructions in a binary can be classified into four types: (i) instructions emitted due to the source code of the binary, (ii) instructions from statically-linked libraries, (iii) nop instructions inserted for alignment, (iv) other instructions inserted by the compiler toolchain, e.g., `_start`. Our system (and thus our ground truth data) currently targets instructions of types (i) and (iii) only. In addition, our current investigation does not consider malware.

3.1 Our Approach

Our system assumes the targeted compiler toolchains are benign (not malicious) but can have bugs. The latter and the possibility of bugs in our own code motivate the need for the automatic checks in §3.3. Figure 1 shows the entire pipeline and the tools used for x86/x64 Linux C programs with clang/gcc/icc. The process is similar on x86/x64 Windows for msvc, except for the file formats and tools.

At a high level, our system starts by using the targeted compiler to generate the assembly files and the symbol table along with the targeted binary. In Linux, these assembly files are further converted into listing files by GNU `as`; in Windows, msvc generates both the assembly and the listing files. The two listing file formats differ, but they both specify a size and an offset for every instruction in every function. We have developed a extractor for each format, and from this point on we will call both types of listing files "LSTs".

From the symbol table, our system retrieves the absolute offset of each function in the binary. By combining this information with the offsets from LSTs, our system generates the absolute offset of every instruction in the binary by adding the absolute offset of its containing function and the relative offset of the instruction within the function, i.e., $Abs\_Insn = Abs\_Func + (Rlt\_Insn - Rlt\_Func)$.

Our reliance on LSTs have both pros and cons. Since LST generation happens to be an existing feature in all our selected compiler toolchains, our approach has the benefit of (i) a symmetry on Linux and Windows, (ii) the potential to support other languages with compiler toolchains that can generate assembly and listing files (of which there are plenty in Linux, though admittedly fewer in Windows), and (iii) the avoidance to depend on compiler instrumentation, which may be impossible for closed-source toolchains. In regards to (iii), we remark that the ability to instrument does not guarantee perfect ground truth. Specifically, in [1], Andriesse et al. marked all ground truth for their Windows (msvc) binaries as fully certain, but they did not do so for the Linux binaries. On the other hand, our LST-based approach has a shortcoming when compared to instrumentation. Specifically, since LSTs are generated before linking, our approach does not support binaries compiled with link-time optimizations (LTO). Although in principle we can instrument the linker, this would not be possible for closed-source toolchains. We remark that [1] did not explain how/if their method differs for their included LTO binaries.

3.2 Implementation Details

In this section, we briefly sketch several selected challenges we met and solved when implementing our ground truth generator. Full detail about them and other challenges can be found in our accompanying tech report on arXiv.

3.2.1 Multiple Encoding Problem. Some x86/x64 assembly instructions, e.g, jmp, can be encoded into different machine code. With respect to our system, multiple encodings occur due to two major reasons. First, both clang and icc use their own internal assemblers. However, since LST generation is supported by only with GNU `as`, we often encounter binaries from clang/icc that disagree with the LSTs generated by GNU `as` due to multiple encodings. Second, our current system design uses (i) a Docker image per OS-compiler pair for the generation of the target binary and its associated intermediate files and (ii) a unified Docker image to process the above data

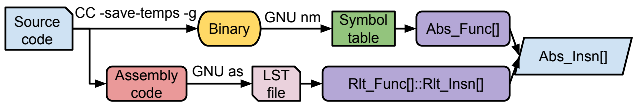

Figure 1: Our disassembly ground truth generation pipeline depicted for Linux clang/gcc/icc. In the Windows equivalent, `cl` directly generates listing files and `dumpbin` replaces `nm`.

to generate the ground truth data. Thus, it is possible for a Linux binary in our suite to be compiled with one version of gcc with its corresponding GNU `as` and yet its LSTs are generated with another version of GNU `as` whose algorithm differs. Our solution is to *iteratively* modify the assembly files by replacing the first mismatched assembly instruction in each function with the encoding obtained from the actual binary and represent it using `.byte` directives. Then we regenerate the LSTs with the modified assembly files and re-extract offsets from the new LSTs. This will either result in full agreement or let us identify other multiple-encoded instructions.

3.2.2 Alignment Directive Translation Problem. With GNU `as`, an assembly file can use alignment directives such as .p2align to specified a desired alignment. Since an alignment directive can be implemented using any combination of nop instructions of different lengths, the alignment instruction(s) inserted into LSTs can differ from those in the binary. Our solution is to introduce to the ground truth representation the concept of "nop regions", which has only an offset and a size but no content. Then, we modify our disassembly accuracy evaluator to act on these regions accordingly.

3.2.3 As-Data Instructions Problem. Although assembly has different syntaxes for instruction and data, compilers may choose to represent an instruction using the data syntax. For example, we have observed icc emitting the instruction `nopl (%rax)` using `.byte` directives. Since these "as-data instructions" are not declared to be instructions in the assembly files and thus the LSTs, without special treatment our ground truth generator would miss them. This would in turn lead to "false false-positives" during disassembly accuracy evaluation. Our solution is to introduce the concept of "optional instructions" in our ground truth representation and use an involved algorithm explained in the tech report to detect and save such instructions into the ground truth after the first pass of ground truth generation. Unfortunately, our current algorithm has a known weakness and we will discuss this in §4.2.

3.3 Correctness Check on Our Ground Truth

To detect potential bugs in both our ground truth generator and the targeted compiler toolchains, we have also built an automatic correctness checker. Recall from §3.2.1, our system iteratively modifies an assembly file (and thus its LST) to determine the byte-length of instructions that have multiple encodings. Our checker verifies that, at the end of the ground truth generation, there is a 1-1 correspondence between every instruction in the final set of LSTs and in the ground truth data. Not counting bugs in our own system, this process has helped us uncover two bugs in GNU `as`. We have reported them and one of them has already been fixed upstream.[1]

[1]https://sourceware.org/bugzilla/show_bug.cgi?id=X, X ∈ {25125, 25621}

Table 1: Projects and Binaries in Our Benchmark Suite

Project	Version	Binary	Linux	Windows
7zip [10]	19.00	7zDec	Y	Y
capstone [5]	4.0.2	cstool	Y	Y
exim [24]	4.86	exim	Y	
lighttpd [12]	1.4.39	lighttpd	Y	
mit-bzip2 [14]	(2006-01-11)	bzip2	Y	Y
mit-gcc [14]	(2006-01-11)	gcc	Y	
mit-gzip [14]	(2006-01-11)	gzip	Y	
mit-oggenc [14]	(2006-01-11)	oggenc	Y	
nginx [7]	1.8.0	nginx	Y	
openssh [16]	7.1p2	sshd	Y	
pcre2 [17]	10.35	pcre2grep	Y	Y
putty [22]	0.73	putty		Y
sqlite [23]	3.30.1	sqlite3	Y	Y
vim [4]	8.2.0821	vim	Y	Y
vsftpd [20]	3.0.3	vsftpd	Y	

4 EVALUATION

In this section, we will first present our benchmark suite (§4.1) and then our evaluation to the following questions:

RQ1: Can our ground truth generator work with various compilers and optimization settings and generate ground truth data on our current benchmark suite? Are there surprises? (§4.2)

RQ2: What are the characteristics of the ground truth data generated using our current benchmark suite? (§4.3)

RQ3: What is the accuracy of the four selected open-source disassemblers according to our ground truth? (§4.4)

4.1 Our Benchmark Suite

Projects. Our current benchmark suite comprises 15 open-source projects and so far we have considered x86 and x64 only. Table 1 shows detailed information of the projects, the selected target binary in a project, and whether a project is included in our Linux / Windows sub-suite. Admittedly our current suite is biased towards Linux due to its history. Its initial composition includes the five Linux programs used in [1] and we added four MIT-produced amalgamations of common Unix programs, one of which happens to be compilable on Windows. On top of these, we added five commonly-recognizable projects that support both Linux and Windows (7zip, capstone, pcre2, sqlite, vim) and one that supports Windows-only (putty). We must caution that we anticipate the membership of our suite will change over time due to community inputs or future investigations. In particular, we believe it would be a very interesting scientific study on how to put together a "best" benchmark suite in view of the competing goals to control the number of included binaries and to increase the complexity exhibited by these binaries.

Toolchains & Settings. On Linux, we support gcc v5.4.0 and v7.5.0, clang v3.8.0 and v6.0.0, and icc v19.1.1.219. For these compilers, we support six settings: `-O0`, `-O1`, `-O2`, `-O3`, `-Ofast`, and `-Os`. On Windows, we support msvc v19.26.28806 with three settings: `/Od`, `/O1`, and `/O2`. The versions of gcc and clang used are the ones distributed in Ubuntu 16.04 LTS and 18.04 LTS and the versions of icc and cl are both the latest as of 2020-07-01. The gcc and clang versions we used are slightly newer than the ones used in [1] because currently we can afford to support only LTS. We leave it as future work to use our ground truth generator on the older compiler versions used in [1] and measure the accuracy of its ground truth.

Table 2: Characteristics of Benchmark Binaries and Ground Truth Data per ISA-OS pair

ISA & OS	x64 Linux	x86 Linux	x64 Windows	x86 Windows
GT Total Size (MB)	2,290	2,280	109	105
Total Binary Size (MB)	980	955	46	46
Max Binary Size (KB)	40,920	19,276	7,463	7,394
Min Binary Size (KB)	58	62	152	152
# Functions	595,917	574,454	37,850	38,449
# Instructions	85,839,363	83,623,983	3,542,001	3,947,966
# Indirect Jumps	319,657	225,273	26,671	25,544
# Distinct Mnemonics	404	389	255	210
Code-Data Interleave	N	N	Y	Y

4.2 Evaluating Our Ground Truth Generator

We have run our ground truth generator over our benchmark suite to ensure compatibility and robustness. Our generator is bundled as a set of programs and Docker images and is written with scripting in mind. Specifically, for each project, our system expects a project directory with specific subdirectory names and specific build scripts at hard-coded locations. Given such a directory, a program of ours will produce a Docker image (for full reproducibility) and run the image to obtain an archive containing the build artifacts in our special git-based format. Finally, we launch another Docker image to process the collected data and produce the ground truth files.

With our current suite, all compilations succeeded with 3 exceptions: icc failed to compile mit-gcc with `-O2`, `-O3`, and `-Ofast` into x86 ELF. The error message is "internal error: 04010022_1238" and we have already reported this bug to Intel.[2] In total, we obtained (i) 420 x64 Linux ELF, (ii) 417 x86 Linux ELF, (iii) 21 x64 Windows COFF, and (iv) 21 x86 Windows COFF. All generated ground truth passed the check described in §3.3.

Unfortunately, our current algorithm to gather "optional instructions" (§3.2.3) can miss an instruction targeted by an indirect jump if the target instruction is encoded as data. When we performed the disassembler evaluation (§4.4), we observed this happening with icc at `-O2`, `-O3`, and `-Ofast`, which accounted for under 1500 and 3000 "false false-positives" in x86 and x64 respectively. So far, this is the only source of error that we are aware of in our ground truth data and we are investigating how to fix this.

4.3 Characteristics of Our Benchmark Suite

Table 2 presents the characteristics of the binaries and their associated ground truth data files in each of the ISA-OS pairs we currently support. The first two rows show the total download sizes for users who trust us and do not want to regenerate the benchmark binaries and the ground truth data from scratch.

Statistics. The binaries in our benchmark suite has a wide range in size—from 58 KB to 40,920 KB. To give a sense of scale/complexity from the perspective of a disassembler, we also counted the number of functions/instructions/indirect jumps and, using llvm-mc 8.0.0, the number distinct mnemonics in these binaries. We stress that the number of indirect jumps is only a proxy to estimate the actual complexity faced by a disassembler because the resolution complexity of an indirect jump can vary greatly.

[2]https://community.intel.com/t5/Intel-C-Compiler/Failed-to-compile-the-MIT-amalgamated-gcc-c-into-a-x86-ELF/m-p/1196443

Code-Data Interleave. The last row of Table 2 was generated by checking whether a linear-sweep disassembler achieves 100% recall and precision against our ground truth. We performed this experiment to repeat part of the study by Andriesse et al. [1] and we arrive at the same conclusion: modern Linux compilers rarely create interleaving code and data. In the machine code due to functions present in the source code of our Linux sub-suite, we did not discover any code-data interleave and therefore a linear-sweep disassembler achieves perfect disassembly when compared with our ground truth. However, we note that code-data interleaves actually exist in some highly-optimized (`-O2`, `-O3`, `-Ofast`) icc-generated x64 binaries in our suite. The reason they do not result in disassembly errors here is because they are due to functions from statically-linked libraries (e.g., `__intel_mic_avx512f_memcpy`) and our ground truth data and our evaluator do not account for instructions in this category. On the other hand, we observed that msvc in both x86 and x64 can generate code-date interleaves where data is present at the end of some functions, including those appearing in the source code. Since linear sweep does not know where a function ends, it outputted many false positives (disassembling data as code) and false negatives (at mismatched instruction boundaries). Finally, we remark that code-data interleaves happen even in modern Linux binaries; see, e.g., [13, Figure 1]. Also, we are aware that code-data interleaves can be common in other ISAs such as ARM. Therefore, we believe continued research in advanced disassembly algorithms (as opposed to settling on linear sweep) is warranted.

4.4 Evaluating Selected Disassemblers

We have used our benchmark suite and our own custom scripts to evaluate four prominent open-source disassemblers: (i) BAP v2.1.0 (2020-05-29), (ii) Ghidra v9.1.2 (2020-02-12), (iii) Radare2 v4.4.0 (2020-04-13), and (iv) ROSE v0.10.4.3 (2020-05-05). While we specifically excluded commercial disassemblers such as IDA Pro and Binary Ninja due to licensing and their limit to API access in their free versions, we believe these vendors can publish their own numbers using our benchmark suite to enable comparisons.

Unstripped Binaries. As strange as it may sound, for this paper we tested all included disassemblers with the *unstripped* version of the binaries from our suite to simulate an experiment where we use stripped binaries but provide each disassembler with perfect function starts. Even though disassembly is arguably more often conducted on stripped binaries and our simulation is not without caveats, there are two reasons behind this decision: (i) After an initial testing with various stripped binaries, we discovered that we do not want to test function start identification (FSI), which as identified in [1] is a much less well-solved challenge in disassembly. For example, the instruction recall of Ghidra on the unstriped x86 vim ELF binary compiled with icc `-O2` is 99.059%, and stripping drops this to 81.429%. (ii) Our current workstation has 128GB of memory and it proves to be insufficient for the stripped experiment without heavily relying on swapping. For example, when running ROSE on the x86 mit-gcc binary compiled by clang 3.8.0 with `-Ofast`, ROSE consumed over 125GB of memory and was soon killed by the OOM-killer when the stack-delta analysis stage was at 88%. We believe an interesting future work would be to provide FSI hints to each disassembler and then also test them with stripped binaries.

Invocations. For fairness, we followed the documentation of each disassembler on how to run it. Even though we recognize that expert users may run a disassembler with various non-default flags and/or third-party plugins, we feel our method better mimics the experience of a typical user. With `${BIN}` denoting an input binary and `${RST}` denoting the output file, our method to run the disassemblers were: (i) BAP: `bap ${BIN} -d asm >${RST}`; (ii) Ghidra: we run `analyzeHeadless` with a Java program we bundled to output all instruction offsets using `currentProgram.getListing().getInstructions(true)` without changing any other configuration; (iii) Radare2: `r2 -Aqc 'pdr @@f >${RST}' ${BIN}`; (iv) ROSE: `rose-recursive-disassemble ${BIN} >${RST}`.

Crashes. We observed a number of crashes in our experiment. In particular, Radare2 seg-faulted on 11 binaries, which we noticed were all large-size binaries including cstool, gcc, nginx, and vim compiled by various compilers. In these cases, we decided to consider the Radare2 output as empty in our accuracy evaluation and we have already filed a bug with the Radare2 developers.[3]

Accuracy. Using our bundled wrappers for the included disassemblers, we collected their outputs on each binary and computed their true positive, false positive, and false negative counts against our ground truth. These are then summarized into the common Receiver Operating Characteristic metrics of precision, recall, and F1 scores. In this paper, we divide the binaries into groups according to the compiler and optimization settings used and by the ISA-OS pair. Since we have 5 compilers and 6 optimization settings in Linux, there are 30 groups in each ISA-Linux pair. The corresponding number is $1 \times 3 = 3$ for each ISA-Windows pair. (For example, each of the 30 x64-Linux groups has 14 binaries; see Table 1.)

To summarize the accuracy of a disassembler in a group, let N be the number of binaries in the group and define the following weights for each binary in the group to adjust for the different number of instructions in each binary:

$$W^{Recall}_{project_i} = \frac{TP_i + FN_i}{\sum_{j=1}^{N} TP_j + FN_j} \qquad W^{Precision}_{project_i} = \frac{TP_i + FP_i}{\sum_{j=1}^{N} TP_j + FP_j}$$

With these weights, we compute the weighted precision, recall, and F1 score for each disassembler over each group in each ISA-OS pair. This in turn allows us to count the number of times when a disassembler has the highest precision/recall/F1 score in each ISA-OS pair, i.e., the number of “wins” in each ISA-OS pair for each metric. These numbers are presented in the left sub-column under each disassembler in Table 3. In addition, to provide a summary of the per-group weighted precision/recall/F1 score, we also compute the harmonic means of the per-group metrics for each disassembler for each ISA-OS pair. These numbers are presented in the corresponding right sub-column.

Adopting F1 scores as our metric for accuracy, ROSE and Ghidra are the most accurate disassemblers for respectively the Linux and Windows binaries in our suite. Overall, every evaluated disassembler achieved a high precision that exceeds 97% in our suite, but the recall is comparably lower. This reflects the common design choice in disassemblers where soundness (every outputted instruction is true) is valued over completeness (every instruction is outputted).

[3]https://github.com/radareorg/radare2/issues/17388

Table 3: #Wins & Harmonic Means of Each ROC Metric of Each Disassembler (Highest F1 in Each ISA-OS Shaded)

Disassemblers		BAP		Ghidra		Radare2		ROSE	
x64 Linux	Recall	0	0.80288	2	0.92657	1	0.83889	27	0.97905
	Prec	4	0.99998	18	0.99965	0	0.99988	8	0.99999
	F1	0	0.89066	2	0.96173	1	0.91234	27	0.98941
x86 Linux	Recall	6	0.82989	6	0.75014	0	0.85280	18	0.97241
	Prec	20	0.99997	5	0.99813	1	0.99994	4	0.99998
	F1	6	0.90703	6	0.85655	0	0.92053	18	0.98601
x64 Win-dows	Recall	0	0.71179	3	0.92938	0	0.77705	0	0.75949
	Prec	0	0.99998	3	1.00000	0	0.99999	0	0.99964
	F1	0	0.83163	3	0.96340	0	0.87454	0	0.86317
x86 Win-dows	Recall	0	0.66079	2	0.95017	0	0.83991	1	0.91219
	Prec	1	0.99965	1	0.99981	1	0.99971	0	0.99905
	F1	0	0.79565	3	0.97436	0	0.91287	0	0.95365

Table 4: Time & Memory Consumption of Each Disassembler Over All 66 **sqlite3 Binaries ({x86, x64}** $\times$ **{ELF, COFF})**

Disassemblers	BAP	Ghidra	Radare2	ROSE
Total User Time (min)	166.65	214.43	47.59	274.96
Total Sys Time (min)	1.41	4.45	0.19	4.97
Total Real Time (min)	168.21	101.03	47.79	65.03
Max Resident Mem (KB)	9,221,200	2,035,936	477,228	4,592,024
Use Multi-core	N	Y	N	Y

Resource Consumption. To compare their performance characteristics, each disassembler invocation was run with `/usr/bin/time -v`. Ideally we would present the measurements with a breakdown by each binary, similar to how SPEC CPU results are typically presented. However, due to space, in this paper we have selected to present with sqlite only. Our suite supports sqlite on all four OS-ISA pairs and there are 66 binaries in total (60 Linux + 6 Windows). We ran each disassembler sequentially on each binary on an otherwise-idle Intel i9-9900K machine with 128GB memory. Table 4 shows the time and memory consumption over the entire sequence. For sqlite, BAP occupied the most amount of memory and ROSE used the most CPU (User+Sys) time. However, Ghidra and ROSE both use multi-core and BAP was in fact slower in wall-clock (Real) time.

5 CONCLUDING REMARKS

In this paper, we proposed to start standardizing the set of binaries used for future disassembler evaluations and presented our work-in-progress benchmark suite. We presented our ground truth generator and evaluated four prominent open-source disassemblers using our ground truth data. Our project is in active development in multiple directions. Aside from fixing the algorithm used in §3.2.3, we believe the most important future work is to investigate what binaries should be included in the benchmark suite. Even limiting to C programs only, we would like to more formally define and increase the complexity captured by the benchmark binaries. For example, while our ground truth representation supports overlapping instructions, none of the programs in our current suite contains this feature. Other future work includes: (i) adding the ability to provide function start hints to each supported disassembler, (ii) adding instructions from statically-linked libraries and data declarations to the ground truth, (iii) adding icc support on Windows, (iv) adding projects from other languages (e.g., C++), and (v) adding other ISAs (e.g., ARM). We sincerely hope the community will provide us with inputs or even pull requests to evolve our benchmark suite into a community standard for future disassembler evaluations.

Acknowledgement. This work was supported by ONR Award No. N00014-17-1-2892. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Office of Naval Research. In addition, we thank the anonymous reviewers for their many suggestions and feedback.

REFERENCES

[1] Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. 2016. An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries. In *Proceedings of the 25th USENIX Security Symposium.* 583–600. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/andriesse

[2] BAP Community. [n.d.]. BinaryAnalysisPlatform/bap: Binary Analysis Platform. https://github.com/BinaryAnalysisPlatform/bap. Online; accessed Jul 1st 2020.

[3] Erick Bauman, Zhiqiang Lin, and Kevin Hamlen. 2018. Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics. In *Proceedings of the 2018 Network and Distributed System Security Symposium.* Internet Society. https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_05A-4_Bauman_paper.pdf

[4] Bram Moolenaar. [n.d.]. welcome home : vim online. https://www.vim.org/. Online; accessed Jul 1st 2020.

[5] Capstone Community. [n.d.]. The Ultimate Disassembly Framework - Capstone - The Ultimate Disassember. https://www.capstone-engine.org/. Online; accessed Jul 1st 2020.

[6] Standard Performance Evaluation Corporation. [n.d.]. SPEC CPU Benchmark Suites. https://www.spec.org/cpu/. Online; accessed Jul 1st 2020.

[7] F5 Networks, Inc. [n.d.]. NGINX | High Performance Load Balancer, Web Server, & Reverse Proxy. https://www.nginx.com/. Online; accessed Jul 1st 2020.

[8] Antonio Flores-Montoya and Eric Schulte. 2020. Datalog Disassembly. In *Proceedings of the 29th USENIX Security Symposium.* USENIX Association, 1075–1092.

[9] R N Horspool and N Marovac. 1980. An approach to the problem of detranslation of computer programs. *Comput. J.* 23, 3 (1980), 223–229. https://doi.org/10.1093/comjnl/23.3.223

[10] Igor Pavlov. [n.d.]. 7-Zip. https://www.7-zip.org/. Online; accessed Jul 1st 2020.

[11] M. Ammar Ben Khadra, Dominik Stoffel, and Wolfgang Kunz. 2016. Speculative Disassembly of Binary Code. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems.* ACM Press, 1–10. https://doi.org/10.1145/2968455.2968505

[12] Lighty Team. [n.d.]. Lighttpd - fly light. https://www.lighttpd.net/. Online; accessed Jul 1st 2020.

[13] Kenneth Miller, Yonghwi Kwon, Yi Sun, Zhuo Zhang, Xiangyu Zhang, and Zhiqiang Lin. 2019. Probabilistic Disassembly. In *Proceedings - International Conference on Software Engineering*, Vol. 2019-May. IEEE, 1187–1198. https://doi.org/10.1109/ICSE.2019.00121

[14] MIT CSAIL. [n.d.]. Large single compilation-unit C programs. https://people.csail.mit.edu/smcc/projects/single-file-programs/. Online; accessed Jul 1st 2020.

[15] NSA. [n.d.]. Ghidra. https://ghidra-sre.org/. Online; accessed Jul 1st 2020.

[16] OpenBSD Foundation. [n.d.]. OpenSSH. https://www.openssh.com/. Online; accessed Jul 1st 2020.

[17] PCRE Community. [n.d.]. PCRE - Perl Compatible Regular Expressions. https://www.pcre.org/. Online; accessed Jul 1st 2020.

[18] radare2 Community. [n.d.]. radare. https://www.radare.org/. Online; accessed Jul 1st 2020.

[19] ROSE Community. [n.d.]. Rose Compiler - Program Analysis and Transformation. http://rosecompiler.org/. Online; accessed Jul 1st 2020.

[20] scarybeasts. [n.d.]. vsftpd - Secure, fast FTP server for UNIX-like systems. https://security.appspot.com/vsftpd.html. Online; accessed Jul 1st 2020.

[21] Benjamin Schwarz, Saumya Debray, and Gregory Andrews. 2002. Disassembly of Executable Code Revisited. In *Proceedings - the 9th Working Conference on Reverse Engineering, WCRE.* IEEE, 45–54. https://doi.org/10.1109/WCRE.2002.1173063

[22] Simon Tatham. [n.d.]. PuTTY: a free SSH and Telnet client. https://www.chiark.greenend.org.uk/~sgtatham/putty/. Online; accessed Jul 1st 2020.

[23] SQLite Consortium. [n.d.]. SQLite Home Page. https://www.sqlite.org/index.html. Online; accessed Jul 1st 2020.

[24] The University of Cambridge. [n.d.]. Exim Internet Mailer. https://exim.org/. Online; accessed Jul 1st 2020.

[25] Richard Wartell, Yan Zhou, Kevin W Hamlen, Murat Kantarcioglu, and Bhavani Thuraisingham. 2011. Differentiating Code from Data in x86 Binaries. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, Vol. 6913 LNAI. Springer, Berlin, 522–536. https://doi.org/10.1007/978-3-642-23808-6_34