An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries

Dennis Andriesse, Xi Chen, and Victor van der Veen, *Vrije Universiteit Amsterdam;*
Asia Slowinska, *Lastline, Inc.;* Herbert Bos, *Vrije Universiteit Amsterdam*

https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/andriesse

This paper is included in the Proceedings of the
25th USENIX Security Symposium
August 10–12, 2016 • Austin, TX

ISBN 978-1-931971-32-4

An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries

Dennis Andriesse[†§], Xi Chen[†§], Victor van der Veen[†§], Asia Slowinska[‡], and Herbert Bos[†§]

[†]{*d.a.andriesse,x.chen,v.vander.veen,h.j.bos*}*@vu.nl*
Computer Science Institute, Vrije Universiteit Amsterdam
[§]*Amsterdam Department of Informatics*
[‡]*asia@lastline.com*
Lastline, Inc.

Abstract

It is well-known that static disassembly is an unsolved problem, but how much of a problem is it in real software—for instance, for binary protection schemes? This work studies the accuracy of nine state-of-the-art disassemblers on 981 real-world compiler-generated binaries with a wide variety of properties. In contrast, prior work focuses on isolated corner cases; we show that this has led to a widespread and overly pessimistic view on the prevalence of complex constructs like inline data and overlapping code, leading reviewers and researchers to underestimate the potential of binary-based research. On the other hand, some constructs, such as function boundaries, are much harder to recover accurately than is reflected in the literature, which rarely discusses much needed error handling for these primitives. We study 30 papers recently published in six major security venues, and reveal a mismatch between expectations in the literature, and the actual capabilities of modern disassemblers. Our findings help improve future research by eliminating this mismatch.

1 Introduction

The capabilities and limitations of disassembly are not always clearly defined or understood, making it difficult for researchers and reviewers to judge the practical feasibility of techniques based on it. At the same time, disassembly is the backbone of research in static binary instrumentation [5, 19, 32], binary code lifting to LLVM IR (for reoptimization or analysis) [38], binary-level vulnerability search [27], and binary-level anti-exploitation systems [1, 8, 29, 46]. Disassembly is thus crucial for analyzing or securing untrusted or proprietary binaries, where source code is simply not available.

The accuracy of disassembly strongly depends on the type of binary under analysis. In the most general case, the disassembler can make very few assumptions on the structure of a binary—high-level concepts like functions and loops have no real significance at the binary level [3]. Moreover, the binary may contain complex constructs, such as overlapping or self-modifying code, or inline data in executable regions. This is especially true for obfuscated binaries, making disassembly of such binaries extremely challenging. Disassembly in general is undecidable [43]. On the other hand, one might expect that compilers emit code with more predictable properties, containing a limited set of patterns that the disassembler may try to identify.

Whether this is true is not well recognized, leading to a wide range of views on disassembly. These vary from the stance that disassembly of benign binaries is a solved problem [48], to the stance that complex cases are rampant [23]. It is unclear which view is justified in a given situation. The aim of our work is thus to study binary disassembly in a realistic setting, and more clearly delineate the capabilities of modern disassemblers.

It is clear from prior work that obfuscated code may complicate disassembly in a myriad of ways [18, 21]. We therefore limit our study to non-obfuscated binaries compiled on modern x86 and x64 platforms (the most common in binary analysis and security research). Specifically, we focus on binaries generated with the popular `gcc`, `clang` and Visual Studio compilers. We explore a wide variety of 981 realistic binaries, including stripped, optimized, statically linked, and link-time optimized binaries, as well as library code that includes handcrafted assembly. We disassemble these binaries using nine state-of-the-art research and industry disassemblers, studying their ability to recover all disassembly primitives commonly used in the literature: instructions, function start addresses, function signatures, Control Flow Graphs (CFG) and callgraphs. In contrast, prior studies focus strongly on complex corner cases in isolation [23, 25]. Our results show that such cases are exceedingly rare, even in optimized code, and that focusing on them leads to an overly pessimistic view on disassembly.

We show that many disassembly primitives can be recovered with better accuracy than previously thought. For instance, instruction accuracy often approaches 100%, even using linear disassembly. On the other hand, we also identify some primitives which are more difficult to recover—most notably, function start information.

To facilitate a better match between the capabilities of disassemblers and the expectations in the literature, we comprehensively study all binary-based papers published in six major security conferences in the last three years. Ironically, this study shows a focus in the literature on rare complex constructs, while little attention is devoted to error handling for primitives that really are prone to inaccuracies. For instance, only 25% of Windows-targeted papers that rely on function information discuss potential inaccuracies, even though the accuracy of function detection regularly drops to 80% or less. Moreover, less than half of all papers implement mechanisms to deal with inaccuracies, even though in most cases errors can lead to malignant failures like crashes.

Contributions & Outline
The contributions of our work are threefold.

(1) We study disassembly on 981 full-scale compiler-generated binaries, to clearly define the true capabilities of modern disassemblers (Section 3) and the implications on binary-based research (Section 4).

(2) Our results allow researchers and reviewers to accurately judge future binary-based research—a task currently complicated by the myriad of differing opinions on the subject. To this end, we release all our raw results and ground truth for use in future evaluations of binary-based research[1].

(3) We analyze the quality of all recent binary-based work published in six major security venues by comparing our results to the requirements and assumptions of this work (Section 5). This shows where disassembler capabilities and the literature are mismatched, and how this mismatch can be resolved moving forward (Section 6).

2 Evaluating Real-World Disassembly

This section outlines our disassembly evaluation approach. We discuss our results, and the implications on binary-based research, in Sections 3–4. Sections 5–6 discuss how closely expectations in the literature match our results.

2.1 Binary Test Suite

We focus our analysis on non-obfuscated x86 and x64 binaries generated with modern compilers. Our experiments are based on Linux (ELF) and Windows (PE) binaries, generated with the popular `gcc` v5.1.1, `clang` v3.7.0 and Visual Studio 2015 compilers—the most recent versions at the time of writing. The x86/x64 instruction set is the most common target in binary-based research. Moreover, x86/x64 is a variable-length instruction set, allowing unique constructs such as overlapping and "misaligned" instructions which can be difficult to disassemble. We exclude obfuscated binaries, as there is no doubt that they can wreak havoc on disassembler performance and we hardly need confirm this in our experiments.

We base our disassembly experiments on a test suite composed of the SPEC CPU2006 C and C++ benchmarks, the widely used and highly optimized `glibc-2.22` library, and a set of popular server applications consisting of `nginx` v1.8.0, `lighttpd` v1.4.39, `opensshd` v7.1p2, `vsftpd` v3.0.3 and `exim` v4.86. This test suite has several properties which make it representative: (1) It contains a wide variety of realistic C and C++ binaries, ranging from very small to large; (2) These correspond to binaries used in evaluations of other work, making it easier to relate our results to the literature; (3) The tests include highly optimized library code, containing handwritten assembly and complex corner cases which regular applications do not; (4) SPEC CPU2006 compiles on both Linux and Windows, allowing a fair comparison of results between `gcc`, `clang`, and Visual Studio.

To study the impact of compiler options on disassembly, we compile the SPEC CPU2006 part of our test suite multiple times with a variety of popular configurations. Specifically: (1) Optimization levels `O0`, `O1`, `O2` and `O3` for `gcc`, `clang` and Visual Studio; (2) Optimization for size (`Os`) on `gcc` and `clang`; (3) Static linking and link-time optimization (`-flto`) on 64-bit `gcc`; (4) Stripped binaries, as well as binaries with symbols. We compile the servers for both x86 and x64 with `gcc` and `clang`, leaving all remaining settings at the Makefile defaults. Finally, we compile `glibc-2.22` with 64-bit `gcc`, to which it is specifically tailored. In total, our test suite contains 981 binaries and shared objects.

2.2 Disassembly Primitives

We test all five common disassembly primitives used in the literature (see Section 5). Some of these go well beyond basic instruction recovery, and are only supported by a subset of the disassemblers we test.

(1) *Instructions*: The pure assembly-level instructions.

(2) *Function starts*: Start addresses of the functions originally defined in the source code.

(3) *Function signatures*: Parameter lists for functions found by the disassembler.

(4) *Control Flow Graph (CFG) accuracy*: The soundness and completeness of the CFG digraphs $G_{cfg} = (V_{bb}, E_{cf})$, which describe how control flow edges $E_{cf} \subseteq V_{bb} \times V_{bb}$ connect the basic blocks V_{bb}. In practice, dis-

[1] `https://www.vusec.net/projects/disassembly/`

assemblers deviate from the traditional CFG; typically by omitting indirect edges, and sometimes by defining a global CFG rather than per-function CFGs. Therefore, we define the *Interprocedural CFG (ICFG)*: the union of all function-level CFGs, connected through interprocedural call and jump edges. This allows us to abstract from the disassemblers' varying CFG definitions, by focusing our measurement on the coverage of basic blocks in the ICFG. We pay special attention to hard-to-resolve basic blocks, such as the heads of address-taken functions and switch/case blocks reached via jump tables.

(5) *Callgraph accuracy*: The correctness of the digraph $G = (V_{cs} \cup V_f, E_{call})$ linking the set V_{cs} of call sites to the function starts V_f through call edges $E_{call} \subseteq V_{cs} \times V_f$. Similarly to the CFG, disassemblers deviate from the traditional callgraph definition by including only direct call edges. In our experiments, we therefore measure the completeness of this *direct callgraph*, considering indirect calls and tailcalls separately in our complex case analysis.

2.3 Complex Constructs

We also study the prevalence in real-world binaries of complex corner cases which are often cited as particularly harmful to disassembly [5, 23, 34].

(1) *Overlapping/shared basic blocks*: Basic blocks may be shared between different functions, hindering disassemblers from properly separating these functions.

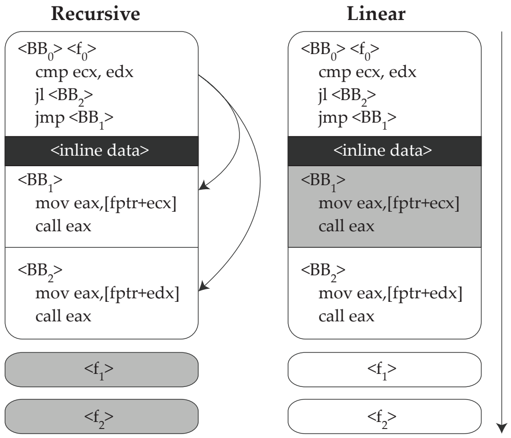
(2) *Overlapping instructions*: Since x86/x64 uses variable-length instructions without any enforced memory alignment, jumps can target any offset within a multi-byte instruction. This allows the same code bytes to be interpreted as multiple overlapping instructions, some of which may be missed by disassemblers.

(3) *Inline data and jump tables*: Data bytes may be mixed in with instructions in a code section. Examples of potential inline data include jump tables or local constants. Such data can cause false positive instructions, and can desynchronize the instruction stream if the last few data bytes are mistakenly interpreted as the start of a multi-byte instruction. Disassembly then continues parsing this instruction into the actual code bytes, losing track of the instruction stream alignment.

(4) *Switches/case blocks*: Switches are a challenge for basic block discovery, because the switch case blocks are typically indirect jump targets (encoded in jump tables).

(5) *Alignment bytes*: Some code (i.e., `nop`) or data bytes may have no semantic meaning, serving only to align other code for optimization of memory accesses. Alignment bytes may cause desynchronization if they do not encode valid instructions.

(6) *Multi-entry functions*: Functions may have multiple basic blocks used as entry points, which can complicate function start recognition.



```
<BB0> <f0>
   cmp ecx, edx
   jl <BB2>
   jmp <BB1>
<BB1>
   mov eax,[fptr+ecx]
   call eax
<BB2>
   mov eax,[fptr+edx]
   call eax
```

Figure 1: Disassembly methods. Arrows show disassembly flow. Gray blocks show missed or corrupted code.

(7) *Tail calls*: In this common optimization, a function ends not with a return, but with a jump to another function. This makes it more difficult for disassemblers to detect where the optimized function ends.

2.4 Disassembly & Testing Environment

We conducted all disassembly experiments on an Intel Core i5 4300U machine with 8GB of RAM, running Ubuntu 15.04 with kernel 3.19.0-47. We compiled our `gcc` and `clang` test cases on this same machine. The Visual Studio binaries were compiled on an Intel Core i7 3770 machine with 8GB of RAM, running Windows 10.

We tested nine popular industry and research disassemblers: *IDA Pro v6.7*, *Hopper v3.11.5*, *Dyninst v9.1.0* [5], *BAP v0.9.9* [7], *ByteWeight v0.9.9* [4], *Jakstab v0.8.4* [17], *angr v4.6.1.4* [36], *PSI v1.1* [47] (the successor of BinCFI [48]), and *objdump v2.22*. ByteWeight yields only function starts, while Dyninst and PSI support only ELF binaries (for Dyninst, this is due to our Linux testing environment). Jakstab supports only x86 PE binaries. We omit angr results for x86, as angr is optimized for x64. PSI is based on objdump, with added error correction. Section 3 shows that PSI (and all linear disassemblers) perform equivalently to objdump; therefore, we group these under the name *linear disassembly*.

All others are recursive descent disassemblers, illustrated in Figure 1. These follow control flow to avoid desynchronization by inline data, and to discover complex cases like overlapping instructions. In contrast, linear disassemblers like objdump simply decode all code bytes consecutively, and may be confused by inline data, possibly causing garbled code like BB_1 in the figure. Recursive disassemblers avoid this problem, but may miss indirect control flow targets, such as f_1 and f_2 in the figure.

2.5 Ground Truth

Our disassembly experiments require precise ground truth on instructions, basic blocks and function starts, call sites, function signatures and switch/case addresses. This information is normally only available at the source level. Clearly, we cannot obtain our ground truth from any disassembler, as this would bias our experiments.

We base our ELF ground truth on information collected by an LLVM analysis pass, and on DWARF v3 debugging information. Specifically, we use LLVM to collect source-level information, such as the source lines belonging to functions and switch statements. We then compile our test binaries with DWARF information, and link the source-level line numbers to the binary-level addresses using the DWARF line number table. We also use DWARF information on function parameters for our function signature analysis. We strip the DWARF information from the binaries before our disassembly experiments.

The line number table provides a full mapping of source lines to binary, but not all instructions correspond directly to a source line. To find these instructions, we use *Capstone v3.0.4* to start a conservative linear disassembly sweep from each known instruction address, stopping at control flow instructions unless we can guarantee the validity of their destination and fall-through addresses. For instance, the target of a direct unconditional jump instruction can be guaranteed, while its fall-through block cannot (as it might contain inline data).

This approach yields ground truth for over 98% of code bytes in the tested binaries. We manually analyze the remaining bytes, which are typically alignment code unreachable by control flow. The result is a ground truth file for each binary test case, that specifies the type of each code byte, as well as instruction and function starts, switch/case addresses, and function signatures.

We use a similar method for the Windows PE tests, but based on information from PDB (Program Database) files produced by Visual Studio instead of DWARF. This produces files analogous to our ELF ground truth format.

We release all our ground truth files and our test suite, to aid in future evaluations of binary-based research and disassembly.

3 Disassembly Results

This section describes the results of our disassembly experiments, using the methodology outlined in Section 2. We first discuss application binaries (SPEC and servers), followed by a separate discussion on highly optimized libraries. Finally, we discuss the impact of static linking and link-time optimization. We release all our raw results, and present aggregated results here for space reasons.

3.1 Application Binaries

This section presents disassembly results for application code. We discuss accuracy results for all primitives, and also analyze the prevalence of complex cases.

3.1.1 SPEC CPU2006 Results

Figures 2a–2e show the accuracy for the SPEC CPU2006 C and C++ benchmarks of the recovered instructions, function starts, function signatures, CFGs and callgraphs, respectively. We show the percentage of correctly recovered (true positive) primitives for each tested compiler at optimization levels `O0`–`O3`. Note that the legend in Figure 2a applies to Figures 2a–2e. All lines are geometric mean results (simply referred to as "mean" from this point); arithmetic means and standard deviations are discussed in the text where they differ significantly. We show separate results for the C and C++ benchmarks, to expose variations in disassembly accuracy that may result from different code patterns.

Some disassemblers support only a subset of the tested primitives. For instance, linear disassembly provides only instructions, and IDA Pro is the only tested disassembler that provides function signatures. Moreover, some disassemblers only support a subset of the tested binary types, and are therefore only shown in the plots where they are applicable. For clarity, the graphs only show results for stripped binaries; our tests with standard symbols (not DWARF information) are discussed in the text.

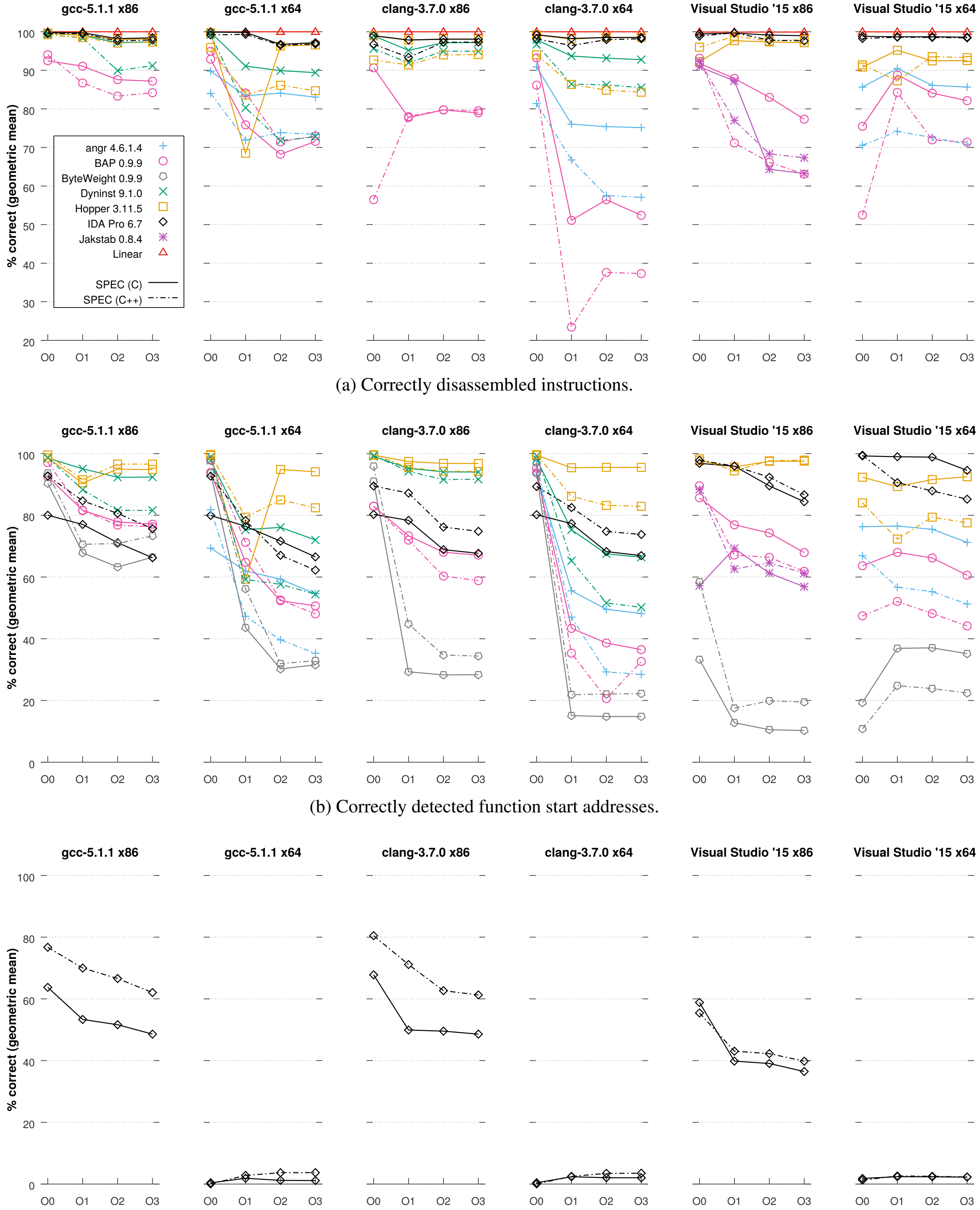
3.1.1.1 Instruction boundaries

Figure 2a shows the percentage of correctly recovered instructions. Interestingly, linear disassembly consistently outperforms all other disassemblers, finding 100% of the instructions for `gcc` and `clang` binaries (without false positives), and 99.92% in the worst case for Visual Studio.

Linear disassembly. The perfect accuracy for linear disassembly with `gcc` and `clang` owes to the fact that these compilers never produce inline data, not even for jump tables. Instead, jump tables and other data are placed in the `.rodata` section.

Visual Studio *does* produce inline data, typically jump tables. This leads to some false positives with linear disassembly (data treated as code), amounting to a worst-case mean of 989 false positive instructions (0.56% of the disassembled code) for the x86 C++ tests at `O3`. The number of missed instructions (false negatives, due to desynchronization) is much lower, at a worst-case mean of 0.09%. This is because x86/x64 disassembly automatically resynchronizes within two or three instructions [21].

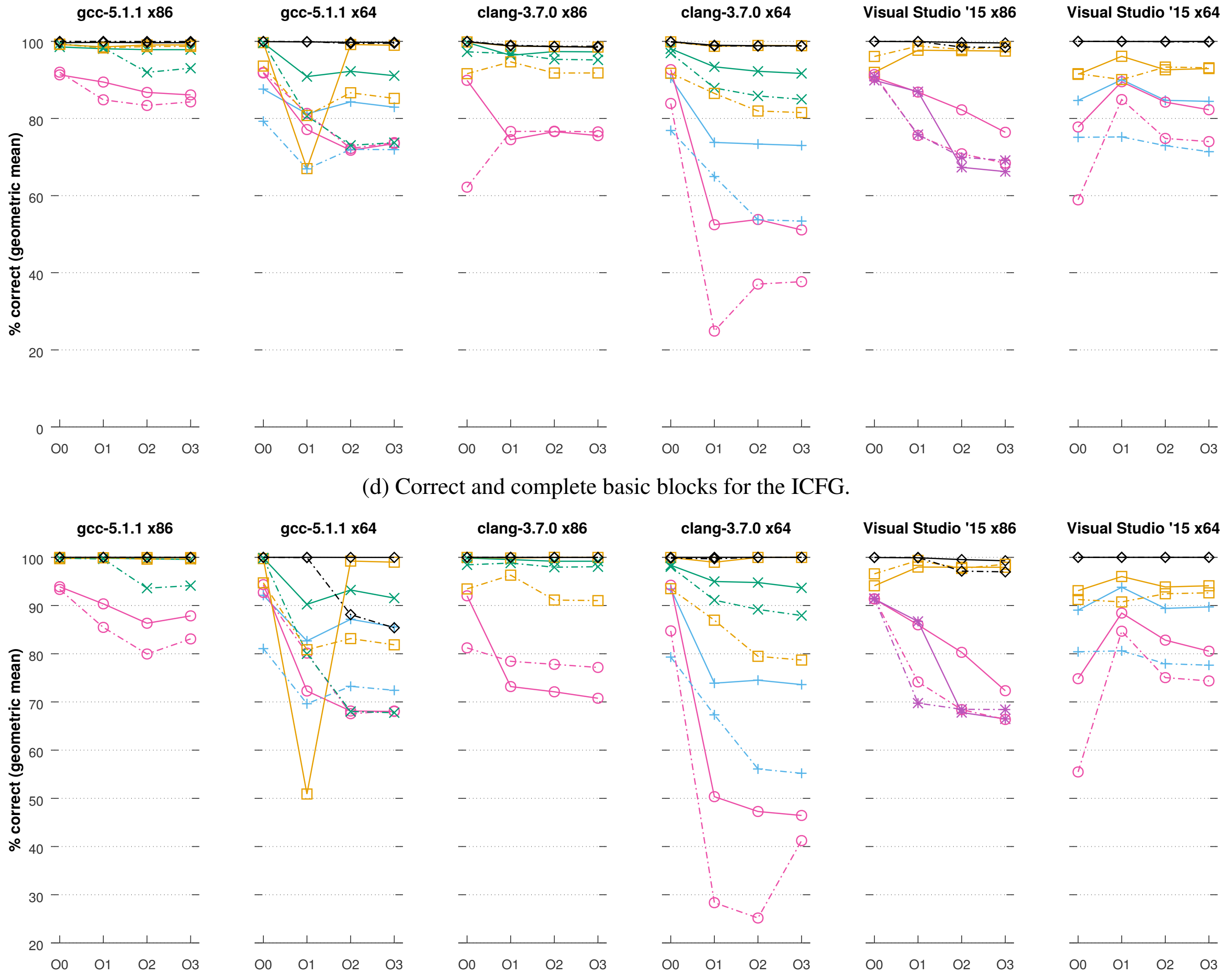
Figure 2: Disassembly results. The legend in Figure 2a applies to Figures 2a–2e. Section 2.4 describes which platforms are supported by each tested disassembler.



(a) Correctly disassembled instructions.

(b) Correctly detected function start addresses.

(c) Correctly detected non-empty function argument lists (IDA Pro only).



(d) Correct and complete basic blocks for the ICFG.

(e) Correctly resolved direct function calls (indirect calls discussed separately).

Recursive disassembly. The most accurate recursive disassembler in terms of instruction recovery is IDA Pro 6.7, which closely follows linear disassembly with an instruction coverage exceeding 99% at optimization levels `O0` and `O1`, dropping to a worst case mean of 96% for higher optimization levels. The majority of missed instructions at higher optimization levels are alignment code for functions and basic blocks, which is quite common in optimized binaries. It consists of various (long) `nop` instructions for `gcc` and `clang`, and of `int 3` instructions for Visual Studio, and accounts for up to 3% of all code at `O2` and `O3`. Missing these instructions is not harmful to common binary analysis operations, such as binary instrumentation, manual analysis or decompilation.

False positives in IDA Pro are less prevalent than in linear disassembly. On `gcc` and `clang`, they are extremely rare, amounting to 14 false positives in the worst test case, with a mean of 0. Visual Studio binaries produce more false positives, peaking at 0.16% of all recovered instructions. Overall, linear disassembly provides the most complete instruction listing, but at a relatively high false positive rate for Visual Studio. IDA Pro finds only slightly fewer instructions, with significantly fewer false positives. These numbers were no better for binaries with symbols.

Dyninst and Hopper achieve best case accuracy comparable to IDA, but not quite as consistently. Some disassemblers, notably BAP, appear to be optimized for `gcc`, and show large performance drops when used on `clang`. The BAP authors informed us that BAP's results depend strongly on the disassembly starting points (i.e., function starts), provided by ByteWeight. We used the default ELF and PE signature files shipped with ByteWeight v0.9.9. Our angr results are based on the CFGFast analysis recommended to us by the angr authors.

Overall, IDA Pro, Hopper, Dyninst and linear disassembly show arithmetic mean results which are extremely close to the geometric means, exhibiting standard deviations below 1%. Other disassemblers have larger standard deviations, typically around 15%, with outliers up to 36% (for BAP on `clang` x86, as visible in Figure 2a).

```
6caf10 <ix86_fp_compare_mode>:
  6caf10:  mov  0x3f0dde(%rip),%eax
  6caf16:  and  $0x10,%eax
  6caf19:  cmp  $0x1,%eax
  6caf1c:  sbb  %eax,%eax
  6caf1e:  add  $0x3a,%eax
  6caf21:  retq
```

Listing 1: False negative indirectly called function for IDA Pro in `gcc`, compiled with `gcc` at `O3` for x64 ELF.

```
480970 <autohelperowl_defendpat156>:
  480970:  push   %rbp
  480971:  push   %r15
  480973:  push   %r14
  480975:  push   %rbx
  480976:  push   %rax
```

Listing 2: False positive function (shaded) for Dyninst, due to misapplied prologue signature, `gobmk` compiled with `clang` at `O1` for x64 ELF.

C versus C++. Accuracy between C and C++ differs most in the lower scoring disassemblers, but the difference largely disappears in the best performing disassemblers. The largest relative difference appears for `clang`.

3.1.1.2 Function starts

The results for function start detection are far more diffuse than those for instruction recovery. Consider Figure 2b, which shows the mean percentage of correctly recovered function start addresses. No one disassembler consistently dominates these results, though Hopper is at the upper end of the spectrum for most compiler configurations in terms of true positives. Dyninst also provides high true positive rates, though not as consistently as Hopper. However, as shown in Figure 3, both Hopper and Dyninst suffer from high false positive rates, with worst case mean false positive rates of 28% and 19%, respectively. IDA Pro provides lower false positive rates of under 5% in most cases (except for x86 Visual Studio, where it peaks at 20%). However, its true positive rate is substantially lower than those of Hopper and Dyninst, regularly missing 20% or more of functions even at low optimization levels. As with instruction recovery, the results for BAP and ByteWeight depend heavily on the compiler configuration, ranging from over 90% accuracy on `gcc` x86 at `O0`, to under 20% on `clang` x64.

Even for the best performing disassemblers, function start identification is far more challenging than instruction recovery. Accuracy drops particularly as the optimization level increases, repeatedly falling from close to 99% true positives at `O0`, to only 82% at `O3`, and worsened by high false positive rates. For IDA Pro, the worst case mean true positive rate is even lower, falling to 62% for C++ on x64 `gcc` at `O3`. Moreover, the standard deviation increases to over 15% even for IDA Pro.

```
8060985:  pop    %ebx
  8060986:  pop    %esi
  8060987:  ret
  8060988:  nop
  8060989:  lea    0x0(%esi,%eiz,1),%esi
```

Listing 3: False positive function (shaded) for Dyninst, due to code misinterpreted as epilogue, `sphinx` compiled with `gcc` at `O2` for x86 ELF.

```
46b990 <Perl_pp_enterloop>:
            [...]
  46ba02:  ja      46bb50 <Perl_pp_enterloop+0x1c0>
  46ba08:  mov     %rsi,%rdi
  46ba0b:  shl     %cl,%rdi
  46ba0e:  mov     %rdi,%rcx
  46ba11:  and     $0x46,%ecx
  46ba14:  je      46bb50 <Perl_pp_enterloop+0x1c0>
            [...]
  46bb47:  pop     %r12
  46bb49:  retq
  46bb4a:  nopw    0x0(%rax,%rax,1)
  46bb50:  sub     $0x90,%rax
```

Listing 4: False positive function (shaded) for Dyninst, due to code misinterpreted as epilogue, `perlbench` compiled with `gcc` at `O3` for x64 ELF.

False negatives. The vast majority of false negatives is caused by indirectly called or tailcalled functions (reached by a `jmp` instead of a `call`), as shown in Listing 1. This explains why the true positive rate drops steeply at high optimization levels, where tail calls and functions lacking standard prologues are common (see Section 3.1.3). Symbols, if available, help greatly in improving accuracy. They are used especially effectively by IDA Pro, which consistently yields over 99% true positives for binaries with symbols, even at higher optimization levels.

False positives. Several factors contribute to the false positive rate. We analyzed a random sample of 50 false positives for Dyninst, Hopper and IDA Pro, the three best performing disassemblers in function detection.

For Dyninst, false positives are mainly due to erroneously applied signatures for function prologues and epilogues. As an example, Listing 2 shows a false positive in Dyninst due to a misidentified prologue: Dyninst scans for the `push %r15` instruction (as well as several other prologue signatures), missing preceding instructions in the function. We observe similar cases for function epilogues. For instance, as shown in Listings 3 and 4, Dyninst assumes a new function following a `ret; nop` instruction sequence. This is not always correct: as shown in the examples, the same code pattern can result from a multi-exit function with padding between basic blocks. Note that both examples could be handled correctly by control flow and semantics-aware disassemblers. In Listing 4, there are intraprocedural jumps towards the basic block at `0x46bb50`, showing that it is not a new function.

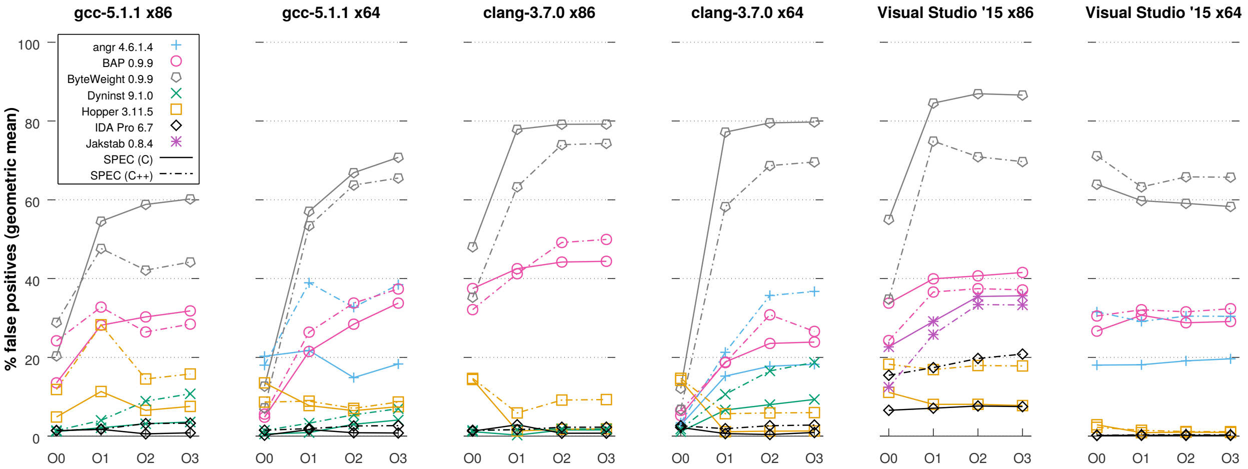


Figure 3: False positives for function start detection (percentage of total detected functions).

```
42cec3:  movss   %xmm0,-0x340(%rbp)
42cecb:  jmpq    42cfc8 <P7PriorifyTransitionVector+0x622>
42ced0:  mov     -0x344(%rbp),%eax
```

Listing 5: False positive function (shaded) for Hopper, due to misclassified switch case block, hmmer compiled with gcc at O0 for x64 ELF.

The false positive in Listing 3 is in effect a nop instruction, emitted for padding by gcc on x86.

All false positives we sampled for Hopper are located directly after padding code, or after a direct jmp (without a fallthrough edge), and are not directly reached by other instructions. An example is shown in Listing 5. Since these instructions are never reached directly, Hopper assumes that they represent function starts. This is not always correct; for instance, the same pattern frequently results from case blocks belonging to switch statements, as seen in Listing 5.

Similarly, the majority of false positives for IDA Pro is also caused by unreachable code assumed to be a new function. However, these cases are far less common in IDA Pro than in Hopper, as IDA Pro more accurately resolves difficult control flow constructs such as switches. Interestingly, the false positive rate for IDA Pro drops to a mean of under 0.3% for x64 Visual Studio 2015. This is because 64-bit Visual Studio uses just one well-defined calling convention, while other compilers use a variety [22].

3.1.1.3 *Function signatures*

Of the tested disassemblers, only IDA Pro supports function signature analysis. Figure 2c shows the percentage of non-empty function argument lists where IDA Pro correctly identified the number of arguments. We focus on non-empty argument lists because IDA Pro defaults to an empty list, skewing our results if counted as correct.

Argument recovery is far more accurate on x86 code, where parameters are typically passed on the stack, than it is on the register-oriented x64 architecture. For x86 code generated by gcc and clang, IDA Pro correctly identifies between 64% and 81% of the argument lists on non-optimized binaries, dropping to 48% in the worst case at O3. Results for Visual Studio are slightly worse, ranging from 36% worst case to 59% in the best case. As for function starts, the standard deviation is just over 15%. On x64 code, IDA Pro recovers almost none of the argument lists, with accuracy between 0.38% and 1.87%.

Performance is significantly better for binaries with symbols, even on x64, but only for C++ code. For instance, IDA Pro's accuracy for gcc x64 increases to a mean of 44% for C++, peaking at 75% correct argument lists. This is because IDA Pro parses mangled function names that occur in C++ symbols, which encode signature information in the function name.

3.1.1.4 *Control Flow Graph accuracy*

Figure 2d presents the accuracy of basic blocks in the ICFG, the union of all function-level CFGs. We found these results to be representative of the per-function CFG accuracy. The accuracy of the ICFG is strongly correlated with instruction discovery; indeed, recursive disassemblers typically find instructions through the process of expanding the ICFG itself. Thus, the disassemblers that perform well in instruction recovery also perform well in CFG construction. For some disassemblers, such as IDA Pro, the basic block true positive rate at high optimization levels even exceeds the raw instruction recovery results (Figure 2a). This is because for the ICFG, we did not count missing nop instructions as false negatives.

IDA Pro consistently achieves a basic block recovery rate of between 98–100%, even at high optimization levels. Even at moderate optimization levels, the results for Hopper and Dyninst are considerably less complete, regularly dropping to 90% or less. For the remaining disassemblers, basic block recovery rates of 75% or less are typical.

All disassemblers except IDA Pro show a considerable drop in accuracy on `gcc` and `clang` for x64, compared to the x86 results. This is strongly correlated with the diminishing instruction and function detection results for these disassembler/architecture combinations (see Figures 2a–2b). This implies that when functions are missed, these disassemblers also fail to recover the instructions and basic blocks contained in the missed functions. In contrast, IDA Pro disassembles instructions even when it cannot attribute them to any function. The difference between x86/x64 and C/C++ results is less pronounced for Visual Studio binaries than for `gcc`/`clang`.

3.1.1.5 Callgraph accuracy

Like ICFG accuracy, callgraph accuracy depends strongly on the completeness of the underlying instruction analysis. As mentioned, the callgraphs returned by the tested disassemblers contain only the direct call edges, and do not deal with address-taken functions. For this reason, Figure 2e presents results for the direct component of the callgraph only. We study the impact of indirect calls on function identification accuracy in our complex case analysis instead (Section 3.1.3). The direct callgraph results in Figure 2e again show IDA Pro to be the most accurate at a consistent 99% function call resolve rate (linking function call edges to function starts), in most cases followed closely by Dyninst and Hopper. This illustrates that the lower accuracy for function starts (Figure 2b) is mainly due to indirectly called functions (such as those called via function pointers or in tail call optimizations).

3.1.2 Server Results

Table 1 shows disassembly results for the servers from our test suite. For space reasons, and because the relative accuracy of the disassemblers is the same as for SPEC, we only show results for IDA Pro, the best overall disassembler. All other results are available externally, as mentioned at the start of Section 3. We compiled all servers for both x86 and x64 with `gcc` and `clang`, using their default Makefile optimization levels.

The server tests confirm that the SPEC results from Section 3.1.1 are representative; all results lie well within the established bounds. As with SPEC, linear disassembly achieved 100% correctness. The `nginx` results warrant closer inspection; given its optimization level `O1`, the function start and argument information is on the low end of the accuracy spectrum. Closer analysis shows that this results from extensive use in `nginx` of indirect calls through function pointers; Section 3.1.1 shows that this negatively affects function information. Indeed, for all tested servers, the accuracy of function start detection is inversely proportional to the ratio of address-taken functions to the total number of instructions. This shows that coding style can carry through the compilation process to have a strong effect on disassembler performance.

	Instructions	Functions	Signatures	ICFG	Callgraph	Instructions	Functions	Signatures	ICFG	Callgraph
			x86					**x64**		
gcc-5.1.1										
`nginx`	99.9	65.5	49.6	100	100	99.9	59.2	0.9	99.9	100
`lighttpd`	99.9	99.5	85.9	99.9	100	99.9	99.5	0.0	99.9	100
`vsftpd`	95.4	93.4	73.6	95.9	99.5	93.0	92.5	4.3	99.9	100
`opensshd`	99.9	86.2	74.9	100	100	99.9	86.2	0.0	100	100
`exim`	99.9	90.1	58.2	99.9	100	99.9	89.9	4.5	99.9	100
clang-3.7.0										
`nginx`	98.5	57.5	44.0	99.5	100	98.6	53.0	0.7	99.4	100
`lighttpd`	98.7	99.5	87.9	99.9	100	99.0	99.5	0.0	99.9	100
`vsftpd`	96.8	93.3	72.9	99.8	100	97.0	92.0	6.6	99.5	99.9
`opensshd`	98.9	86.5	78.1	100	100	99.2	86.3	0.0	100	100
`exim`	99.0	82.7	54.6	99.3	100	99.1	81.7	5.4	99.4	100

Table 1: IDA Pro 6.7 disassembly results for server tests (% correct, per test case).

3.1.3 Prevalence of Complex Constructs

Figure 4 shows the prevalence of complex constructs in SPEC CPU2006, which pose special disassembly challenges. We also analyzed these constructs in the server binaries, finding no significantly different results.

We did not encounter any overlapping or shared basic blocks in either the SPEC or server tests on any compiler. This is surprising, as these constructs are frequently cited in the literature [5, 17, 23]. Closer inspection showed that all the cited cases of overlapping blocks are due to constructs which we classify more specifically, namely overlapping instructions and multi-entry functions. These constructs are exceedingly rare, and occur almost exclusively in library code (discussed in Section 3.2.2). This finding fits with the examples seen in the literature, which all stem from library code, most commonly `glibc`.

No overlapping instructions occur in Linux application code, and only a handful in Windows code (with a mean of zero, and a maximum of 3 and 10 instructions for x86 and x64 Visual Studio, respectively). Multi-entry functions are somewhat more common. All cases we found consisted of functions with optional basic blocks that can execute before the main function body, and finish by jumping over the main function body prologue. Figure 4 lists such jumps as *multi-entry jumps*, and shows

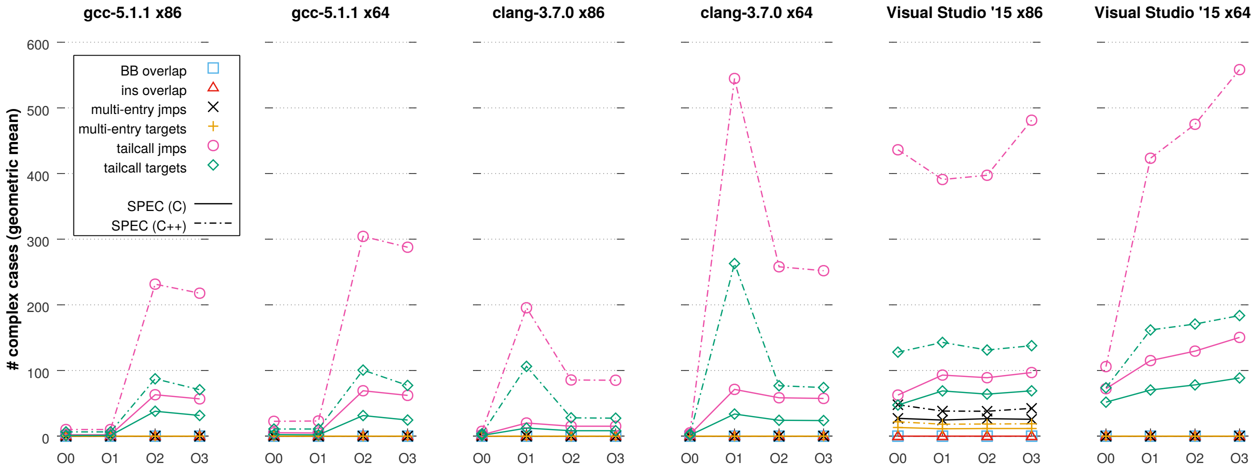


Figure 4: Prevalence of complex constructs in SPEC CPU2006 binaries.

the targeted main function bodies as *multi-entry targets*. In binaries compiled with gcc and clang, we found up to 18 multi-entry jumps for C code, and up to 64 for C++, with the highest prevalence in x64 binaries. Visual Studio produced up to 172 multi-entry jumps for C, and up to 88 for C++, the construct being most prevalent in x86 code. This kind of multi-entry function is handled well by disassemblers in practice, producing no notable decrease in disassembly accuracy compared to other functions.

Tailcalls form the most prevalent complex case, and do negatively affect function start detection if the target function is never called normally (see Section 3.1.1). The largest number of tailcalls (listed as *tailcall jumps* in Figure 4) is found in clang x64 C++ binaries, at a mean of 545 cases. Visual Studio produces a similar number of tailcalls. For clang, the number of tailcalls peaks at optimization level O1, while Visual Studio peaks at O3. For clang (and to a lesser extent gcc), higher optimization levels can lead to a decrease in tailcalls through other modifications like code merging and code elimination.

Jump tables (due to switches) are by far the most common case of inline data. They occur as inline data only on Visual Studio (gcc and clang place jump tables in the .rodata section). As seen in Section 3.1.1, inline data causes false positive instructions especially in linear disassembly (peaking at 0.56% false positives).

Another challenge due to jump tables is locating all case blocks belonging to the switch; these are typically reached indirectly via a jump that loads its target address from the jump table. Linear disassembly covers 100% of case blocks correctly on gcc and clang (see Section 3.1.1), and also achieves very high accuracy for Visual Studio. The best performing recursive disassemblers, most notably IDA Pro, also achieved very high coverage of switch/case blocks; coverage of these blocks is comparable to the overall instruction/basic block recovery rates. This is because many recursive disassemblers have special heuristics for identifying and parsing standard jump tables.

3.1.4 Optimizing for Size

At optimization levels O0–O3, no overlapping or shared basic blocks occur. A reasonable hypothesis is that compilers might more readily produce such blocks when optimizing for size (optimization level Os) rather than for performance. To verify this, we recompiled the SPEC C and C++ benchmarks with size optimization, and repeated our disassembly tests.

Even for size-optimized binaries, we did not find any overlapping or shared blocks. Moreover, the accuracy of the instruction boundaries, callgraph and ICFG did not significantly differ from our results for O0–O3. Function starts and argument lists were comparable in precision to those for performance-optimized binaries (O2–O3).

3.2 Shared Library Objects

This section discusses our disassembly results and complex case analysis for library code. Libraries are often highly optimized, and therefore contain more complex (handcrafted) corner cases than application code. We focus our analysis on glibc-2.22, the standard C library used in GNU systems, compiled in its default configuration (gcc with optimization level O2). This is one of the most widespread and highly optimized libraries, and is often cited as a highly complex case [5, 23].

3.2.1 Disassembly Results

Table 2 shows disassembly results for glibc-2.22, for all tested disassemblers that support 64-bit ELF binaries. Nearly all disassemblers display significantly lower

	Instructions	Functions	Signatures	ICFG	Callgraph
gcc-5.1.1 x64					
angr	64.4	75.6	—	70.2	87.9
BAP	65.3	79.6	—	72.4	84.8
ByteWeight	—	29.3	—	—	—
Dyninst	79.7	85.2	—	87.6	95.5
Hopper	84.3	93.3	—	90.6	93.9
IDA Pro	96.0	92.0	5.4	99.9	99.9
Linear	99.9	—	—	—	—

Table 2: Disassembly results for `glibc` (% correct).

```
7b05a:  cmpl          $0x0,%fs:0x18
7b063:  je            7b066
7b065:  lock cmpxchg %rcx,0x3230fa(%rip)
```

Listing 6: Overlapping instruction in `glibc-2.22`.

```
e9a30 <splice>:
  e9a30:  cmpl   $0x0,0x2b9da9(%rip)
  e9a37:  jne    e9a4c <__splice_nocancel+0x13>
e9a39 <__splice_nocancel>:
  e9a39:  mov    %rcx,%r10
  e9a3c:  mov    $0x113,%eax
  e9a41:  syscall
  e9a43:  cmp    $0xfffffffffffff001,%rax
  e9a49:  jae    e9a7f <__splice_nocancel+0x46>
  e9a4b:  retq
  e9a4c:  sub    $0x8,%rsp
  e9a50:  callq  f56d0 <__libc_enable_asynccancel>
  [...]
```

Listing 7: Multi-entry function in `glibc-2.22`.

accuracy on instruction boundaries than the mean for application binaries in equivalent compiler configurations. Only IDA Pro and linear disassembly are on par with their performance on application code, achieving very good accuracy without any false positives. Note that objdump achieves 99.9% accuracy instead of the usual 100% for ELF binaries. This is because unlike IDA Pro, it does not explicitly separate the overlapping instructions that occur in `glibc` (see Section 3.2.2).

Function start results are on par with, or even exceed the mean for application binaries; this holds true for all disassemblers. Moreover, the accuracy of function argument lists (5.4%) is much higher than one would expect from the x64 SPEC CPU2006 results (under 1% accuracy). This is because IDA Pro comes with a set of code signatures designed to recognize standard library functions that are statically linked into binaries.

For the ICFG, we see the same pattern as for instructions: all disassemblers perform worse than for application code, while IDA Pro delivers comparable accuracy. Callgraph accuracy is below the mean for most disassemblers, though IDA Pro and Dyninst perform very close to the mean, and BAP well exceeds it.

3.2.2 Complex Constructs

Overall, we found the `glibc-2.22` code to be surprisingly well-behaved. Our analysis found no overlapping or shared basic blocks, and no inline data. Indeed, the `glibc` developers have taken special care to prevent this, explicitly placing data and jump tables in the `.rodata` section even when manually declared in handwritten assembly code. Prior work has analysed earlier versions of `glibc`, showing that inline jump tables *are* present in `glibc-2.12` [23]. Moreover, inline zero-bytes used for function padding are confirmed in versions up to 2.21. This is worth noting, as older `glibc` versions may still be encountered in practice. Our analysis of `glibc` versions ranging from 2.12 to 2.22 shows consistently improving disassembler-friendliness over time.

We did find some complex constructs that do not occur in application code, the most notable being overlapping instructions. We found 31 such instructions in `glibc`. All of these are instructions with optional prefixes, such as the one shown in Listing 6. These overlapping instructions are defined manually in handcrafted assembly code, and typically use a conditional jump to optionally skip a `lock` prefix. They correspond to frequently cited complex cases in the literature [5, 23].

In addition, we found 508 tailcalls resulting from the compiler's normal optimization; a number comparable to application binaries of similar size as `glibc`. We also found significantly more multi-entry functions than in the SPEC benchmarks. Most of these belong to the `_nocancel` family, explicitly defined in `glibc`, an example of which is shown in Listing 7. These functions provide optional basic blocks which can be prefixed to the main function body to choose a threadsafe variant of the function. These prefix blocks end by jumping over the prologue of the main function body, a pattern also sometimes seen in application code.

Given that all non-standard complex constructs in `glibc` are due to handwritten assembly, we manually analyzed all assembly code in `libc++` and `libstdc++`. However, the amount of assembly in these libraries is very limited and revealed no new complex constructs. This suggests that the optimization constructs in `glibc` are typical for low-level libraries, and less common in higher-level ones such as the C++ standard libraries.

3.3 Static Linking & Linker Optimization

Static linking can reduce disassembler performance on application binaries by merging complex library code into the binary. Link-time optimization performs intermodular optimization at link-time, as opposed to more local compile-time optimizations. It is a relatively new feature that is gaining in popularity, and could worsen disassembler performance if combined with static linking, by optimizing application and library code as a whole. To study

	Instructions	Functions	Signatures	ICFG	Callgraph
gcc-5.1.1 x64 with -static					
SPEC/C `O0`	96.2	69.4	0.1	98.3	98.2
SPEC/C `O1`	96.2	68.4	0.2	98.6	98.4
SPEC/C `O2`	95.5	67.1	0.2	98.8	98.9
SPEC/C `O3`	95.6	65.7	0.2	98.7	98.7
SPEC/C `Os`	95.9	67.8	0.2	98.7	98.4
gcc-5.1.1 x64 with -static and -flto					
SPEC/C `O0`	96.3	69.3	0.2	98.5	98.3
SPEC/C `O1`	96.0	68.6	0.3	98.6	98.4
SPEC/C `O2`	95.0	67.4	0.3	98.3	98.0
SPEC/C `O3`	95.2	66.9	0.3	98.3	98.4
SPEC/C `Os`	95.5	67.8	0.2	98.4	97.7

Table 3: IDA Pro 6.7 disassembly results for static and link-time optimized SPEC C benchmarks (% correct, geometric mean).

the effects of these options, we recompiled the SPEC CPU2006 C benchmarks, statically linking them with `glibc-2.22` using `gcc`'s `-static` flag. Subsequently, we repeated the process with both static linking and link-time optimization (`gcc`'s `-flto`) enabled.

As expected, static linking merges complex cases from `glibc` into SPEC, including overlapping instructions. The effect on disassembly performance is shown in Table 3 for IDA, the overall best performing disassembler in our `glibc` tests. The impact is slight but noticeable, with an instruction accuracy drop of up to 3 percentage points compared to baseline SPEC; about the same as for `glibc`. As can be seen in Table 3, link-time optimization does not significantly decrease disassembly accuracy compared to static linking only.

Function start detection suffers from static linking mostly at lower optimization levels, dropping from a mean of 80% to just under 70% for `O0`; at level `O3` the performance is not significantly reduced. Again, link-time optimization does not worsen the situation compared to pure static linking. For the ICFG and callgraph tests, a small accuracy drop is again seen at lower optimization levels, again with no more adverse effects due to link-time optimization. For instance, ICFG accuracy drops from close to 100% mean in baseline SPEC to just over 98% in statically linked SPEC at `O0`, while the results at `O2` and `O3` show no negative impact. We suspect that this is a result of optimized library code being linked in even at lower optimization levels. Overall, we do not expect any significant adverse impact on binary-based research as link-time optimization gains in popularity.

4 Implications of Results

This section discusses the implications of our results for three popular directions in binary-based research: (1) Control-Flow Integrity, (2) Decompilation, and (3) Automatic bug search. A detailed comparison of our results to assumptions in the literature is given in Section 5.

4.1 Control-Flow Integrity

Control-Flow Integrity (CFI) is currently one of the most popular research directions in systems security, as shown in Table 6. Binary-level CFI typically relies on binary instrumentation to insert control flow protections into proprietary or legacy binaries [1, 10, 24, 29, 41, 45, 46, 48]. Though a wide variety of CFI solutions has been proposed, most of these have similar binary analysis requirements, due to their common aim of protecting indirect jumps, indirect calls, and return instructions. We structure our discussion around what is needed to analyze and protect each of these control edge types.

Indirect calls. Typically, protecting an indirect call requires instrumenting both the call site (the `call` instruction itself, possibly including parameters), and the call target (the called function). Finding call sites relies mainly on accurate and complete disassembly of the basic instructions. As shown in Figure 2a, these can be recovered with extremely high accuracy, even 100% accuracy for linear disassembly on `gcc` and `clang` binaries. Thus, a binary-level CFI solution is unlikely to encounter problems analyzing and instrumenting call sites.

For Visual Studio binaries, there is a chance that a small percentage of call sites may be missed. Depending on the specific CFI solution, it may be possible to detect calls from uninstrumented sites in the target function, triggering a runtime error handling mechanism (see Section 5). Since these cases are rare, it is then feasible to perform more elaborate (slow path) alternative security checks.

The main challenge is to accurately detect all possible target functions for each indirect call. As a basic prerequisite, this requires finding the complete set of indirectly called functions. As shown in Section 3.1.1 and Figure 2b, this is one of the most challenging problems in disassembly — at high optimization levels, 20% or more of all functions are routinely missed.

Moreover, fine-grained CFI systems must perform even more elaborate analysis to decide which functions are legal targets for each indirect call site. Overestimating the set of legal targets leads to attacks which redirect indirect calls to unexpected functions [12]. Matching call sites to a set of targets typically requires an accurate (I)CFG, so that control-flow and data-flow analysis can be performed to determine which function pointers are passed to each call site. Figure 2d and Sections 3.1.1–3.1.3 show that an accurate and complete ICFG is typically available, including accurate resolution of switch/jump tables in the best disassemblers. Although this type of analysis remains extremely challenging, especially if done interprocedurally (requiring accurate indirect call resolution), it is at least not limited by the accuracy of basic blocks or direct control edges.

Additionally, fine-grained CFI systems can benefit from function signature information, to further narrow down the set of targets per call site by matching the function prototype to parameters passed at the call site [39]. Though signature information is often far from complete (Figure 2c), especially on x64, the information which is available can still be useful — even with incomplete information, the target set can be reduced, directly leading to security improvements. However, care must be taken to make the analysis as conservative as possible; if this is not done, the inaccuracy of function signature information can easily cause illegal function calls to be allowed, or worse, can cause legal calls to be inadvertently blocked.

Indirect jumps. Protecting indirect jumps requires analysis similar to the requirements for indirect calls. However, as indirect jumps are typically intraprocedural, protecting them usually does not rely on function detection. Instead, accurate switch/jump table resolution is required, which is available in disassemblers like IDA Pro (Section 3.1.3).

Return instructions. Return instructions are typically protected using a shadow stack, which requires instrumenting all call and return sites (and jumps, to handle tailcalls) [8]. Given the accurate instruction recovery possible with modern disassemblers (Figure 2a), it is possible to accurately and completely instrument these sites.

Summarizing, the main challenge for modern CFI lies in accurately and completely protecting indirect call sites. The reasons for this are twofold: (1) Function detection is one of the most inaccurate primitives (especially for indirectly called functions), even in state of the art disassemblers, and (2) It is currently very difficult to recover rich information, such as function signature information, through disassembly. This makes it extremely challenging to accurately couple indirect call sites with valid targets.

4.2 Decompilation

Instead of translating a binary into assembly instructions, decompilers lift binaries to a higher-level language, typically (pseudo-) C. Decompilers are typically built on top of a disassembler, and therefore rely heavily on the quality of the disassembly [33, 44].

As most decompilers operate at function granularity, they rely on accurate function start information. Moreover, they must translate all basic blocks belonging to a function, requiring knowledge of the function's CFG. In effect, this requires not only accurate function start detection, but accurate function boundary detection. As described in related work, function boundary detection is even more challenging than function start detection, as it additionally requires locating the end address of each function [4]. This is difficult, especially in optimized binaries, where tailcalls often blur the boundaries between functions (since the `jmp` instructions used in tailcalls can easily be mistaken for intraprocedural control transfers).

In addition to function detection, decompilers rely on accurate instruction disassembly, and can also greatly benefit from function signature/type information. Moreover, switch detection is required to correctly attribute all switch case blocks to their parent function. Finally, callgraph information is useful to understand the connections between decompiled functions.

The impact of inaccuracies for decompilation is not as severe as for CFI systems, since decompiled code is typically intended for use in manual reverse engineering rather than automated analysis. However, disassembly errors can still affect the decompilation process itself, especially in later passes (such as stack frame analysis or data type analysis passes) over the raw decompiled function. Such analysis phases, as well as human reverse engineers, must take into account the high probability of errors in function boundary and signature information.

4.3 Automatic Bug Search

The binary analysis requirements of automatic bug search systems depend on the type of bug being searched for, and the granularity of the search. In practice, many such systems operate at the function level, both for ease of analysis, and because it is a suitable search-granularity for common bugs, such as stack-based bugs [14, 27, 50]. Operating at the function level is also useful for interoperability with other binary analysis primitives, such as symbolic execution, which are powerful tools for semantic analysis but do not scale to full binaries [14].

Thus, like decompilation, many automatic bug search systems rely on accurate function boundary information and per-function CFGs. Fortunately, despite the relatively large inaccuracies in the input information, the output of bug detection systems tends to degrade gracefully — input inaccuracies may lead to bugs being missed, but typically do not affect the correctness of the analysis for other parts of the code. Quantifying the accuracy of the inputs (disassembly, CFG, function boundaries) helps users to determine the expected output completeness of automatic bug search systems.

5 Disassembly in the Literature

Given our disassembly results, we studied recent binary-based research to determine how well the capabilities

	# Papers	Instructions	Functions	Signatures	CFG	Callgraph
angr	0	0	0	0	0	0
BAP	2	1	2	1	2	0
ByteWeight	0	0	0	0	0	0
Dyninst	1	1	0	0	1	1
Hopper	0	0	0	0	0	0
IDA Pro	13	11	6	2	11	4
Jakstab	0	0	0	0	0	0
PSI/BinCFI	4	3	3	0	3	2
Linear	2	2	1	0	1	1
Other/Custom	8	7	2	0	6	3
Total	30	25	14	3	24	11

Table 4: Primitives/disassemblers used in the literature.

		All papers		Top-tier	
Property	**Subproperty**	**#**	**%**	**#**	**%**
Windows PE x86/x64 (16 papers, 12 top-tier)					
Obfuscated code		5	31%	4	33%
Optimized binaries		14	88%	11	92%
Stripped binaries		15	94%	11	92%
Recursive disassembly		16	100%	12	100%
Needs relocation info		2	12%	2	17%
Primitive errors discussed	Instructions	5 (13)	38%	5 (9)	56%
	Functions	1 (5)	20%	1 (4)	25%
	Signatures	0 (2)	0%	0 (2)	0%
	Callgraph	4 (5)	80%	4 (5)	80%
	CFG	5 (13)	38%	5 (10)	50%
Complex cases discussed		5	31%	5	42%
Primitive errors handled	Overestimate	4	25%	4	33%
	Underestimate	3	19%	2	17%
	Runtime	1	6%	1	8%
Errors are fatal		13	81%	11	92%
Linux ELF x86/x64 (14 papers, 10 top-tier)					
Obfuscated code		1	7%	1	10%
Optimized binaries		13	93%	9	90%
Stripped binaries		11	79%	7	70%
Recursive disassembly		12	86%	8	80%
Primitive errors discussed	Instructions	6 (12)	50%	6 (9)	67%
	Functions	3 (9)	33%	3 (6)	50%
	Signatures	1 (1)	100%	1 (1)	100%
	Callgraph	2 (6)	33%	2 (4)	50%
	CFG	5 (11)	45%	5 (8)	62%
Complex cases discussed		1	7%	1	10%
Primitive errors handled	Overestimate	4	29%	3	30%
	Underestimate	0	0%	0	0%
	Runtime	1	7%	1	10%
Errors are fatal		8	57%	6	60%

Table 5: Properties of binary-based papers (number and percentage of papers). Numbers in parentheses indicate the total number of papers that use this primitive.

of disassemblers match the expectations in the literature. Our study covers research published between 2013 and 2015 in all top-tier systems security conferences, namely S&P (Oakland), CCS, NDSS and USENIX Security. In addition, we cover research published in these same years at RAID and ACSAC, two other major conferences which are popular targets for such research.

We found 30 papers on binary-based research published in these venues, summarized in Table 6. The rest of this section presents aggregated findings to provide a degree of anonymization for these papers.

Table 4 shows the primitives and disassemblers used in these papers. IDA Pro is by far the most popular, for all primitives; our disassembly results (Section 3) justify this choice. Despite its good accuracy, linear disassembly is among the least used, even for papers that handle only ELF binaries. This may result from the widespread belief that inline data causes far more problems than we found.

Instructions are the most often needed primitive, used by 25 of the 30 papers. It is followed by the CFG (24 papers) and function starts (14 papers). Function signature information is needed by only 3 of the analyzed papers. One paper used linear disassembly as a basis for building a CFG and callgraph, and scanning for function starts.

Table 5 provides a more detailed insight into the properties of the papers we analyzed. We distinguish between papers that target Windows PE binaries, and those that target Linux ELF. This is because some complex cases, such as inline data, are more often generated by Visual Studio, deserving closer attention in Windows papers.

Most papers that support obfuscated binaries target Windows (33% of papers versus 10% for Linux). This is because obfuscation typically occurs in malware, which is more prevalent on Windows. Though we do not consider obfuscated binaries in our tests, it is still interesting to know how many papers target such binaries. After all, these papers should pay special attention to disassembly errors and complex corner cases. Unfortunately, this is not the case; only 50% of papers that support obfuscation discuss potential errors, while 33% implement error handling. This is no better than the overall number. Moreover, only 17% of these papers explicitly discuss complex cases; far below the overall rate for Windows.

Nearly all papers support optimized binaries (90% or more for both Linux and Windows, overall as well as top-tier). Stripped binaries are supported by an equally large majority of papers on Windows, and by a slightly smaller majority on Linux. Curiously, the number of top-tier papers that support stripped binaries on Linux (70%) is significantly less than the overall number (79%).

The vast majority of papers use recursive disassembly (100% on Windows and 86% on Linux), with IDA Pro being the most popular disassembler. The few papers that do use linear disassembly are based on objdump, and augment it with a layer of error correction. Interestingly, these papers claim perfect (100% accurate) or close to perfect disassembly. As shown in Section 3.1.1, this precision on Linux binaries owes entirely to the core linear disassembly, making any error correction redundant other than for a few corner cases in library code (and obfuscated code, which these papers do not consider).

A relatively small percentage of Windows papers use relocation information to find disassembly starting points. At 17%, this number is slightly higher for top-tier papers.

Discussion on disassembly errors and complex cases is somewhat lacking in the analyzed papers. For most prim-

Title	Authors	Venue	Year	Top-tier
A Principled Approach for ROP Defense [30]	Qiao et al.	ACSAC	2015	
Binary Code Continent: Finer-Grained Control Flow Integrity (...) [41]	Wang et al.	ACSAC	2015	
Blanket Execution: Dynamic Similarity Testing for Program (...) [11]	Egele et al.	USENIX Sec	2014	✓
BYTEWEIGHT: Learning to Recognize Functions in Binary Code [4]	Bao et al.	USENIX Sec	2014	✓
CoDisasm: Medium Scale Concatic Disassembly of Self-Modifying (...) [6]	Bonfante et al.	CCS	2015	✓
Control Flow and Code Integrity for COTS binaries [49]	Zhang et al.	ACSAC	2015	
Control Flow Integrity for COTS Binaries [48]	Zhang et al.	USENIX Sec	2013	✓
Cross-Architecture Bug Search in Binary Executables [27]	Pewny et al.	S&P	2015	✓
DUET: Integration of Dynamic and Static Analyses for Malware (...) [15]	Hu et al.	ACSAC	2013	
Dynamic Hooks: Hiding Control Flow Changes within (...) [40]	Vogl et al.	USENIX Sec	2014	✓
Hardware-Assisted Fine-Grained Control-Flow Integrity (...) [10]	Davi et al.	RAID	2015	
Heisenbyte: Thwarting Memory Disclosure Attacks using (...) [37]	Tang et al.	CCS	2015	✓
High Accuracy Attack Provenance via Binary-based (...) [20]	Hyung Lee et al.	NDSS	2013	✓
Improving Accuracy of Static Integer Overflow Detection in Binary [50]	Zhang et al.	RAID	2015	
Leveraging Semantic Signatures for Bug Search in Binary Programs [28]	Pewny et al.	ACSAC	2014	
Native x86 Decompilation Using Semantics-Preserving (...) [33]	Schwartz et al.	USENIX Sec	2013	✓
No More Gotos: Decompilation Using Pattern-Independent (...) [44]	Yakdan et al.	NDSS	2015	✓
Opaque Control-Flow Integrity [24]	Mohan et al.	NDSS	2015	✓
Oxymoron Making Fine-Grained Memory Randomization Practical (...) [2]	Backes et al.	USENIX Sec	2014	✓
Practical Context-Sensitive CFI [1]	Andriesse et al.	CCS	2015	✓
Practical Control Flow Integrity & Randomization for (...) [46]	Zhang et al.	S&P	2013	✓
Reassembleable Disassembling [42]	Wang et al.	USENIX Sec	2015	✓
Recognizing Functions in Binaries with Neural Networks [35]	Chul et al.	USENIX Sec	2015	✓
ROPecker: A Generic and Practical Approach for Defending (...) [9]	Cheng et al.	NDSS	2014	✓
StackArmor: Comprehensive Protection from Stack-based (...) [8]	Chen et al.	NDSS	2015	✓
Towards Automated Integrity Protection of C++ Virtual Function (...) [13]	Gawlik et al.	ACSAC	2014	
Towards Automatic Software Lineage Inference [16]	Jang et al.	USENIX Sec	2013	✓
vfGuard: Strict Protection for Virtual Function Calls (...) [29]	Prakash et al.	NDSS	2015	✓
VTint: Protecting Virtual Function Tables' Integrity [45]	Zhang et al.	NDSS	2015	✓
X-Force: Force-Executing Binary Programs for Security (...) [26]	Peng et al.	USENIX Sec	2014	✓

Table 6: Set of papers discussed in the literature study.

itives on Windows, at best 50% of papers discuss what happens if the primitive is not recovered perfectly. This number applies to the top-tier papers; overall, the number is even lower. The number for Linux-based papers is slightly better, though even here only a small majority of papers devote significant attention to potential problems. One would expect more thorough discussion, especially given that between 80% and 90% of Windows papers, and around 60% of Linux papers, may suffer malignant failures given imperfect primitives. The issue is most apparent in the Windows papers that require function start information. Only 25% of the top-tier papers that require function starts consider potential errors in this information, even though Section 3.1.1 shows that function starts are quite challenging to recover accurately.

The percentage of Windows papers that discuss complex cases such as inline data varies from 31% overall to 42% for top-tier papers. Again, this is less than we would expect given the prevalence of inline jump tables generated by Visual Studio. The number for papers that target Linux is even lower, though this causes fewer issues as complex cases in ELF binaries are rare.

There is a strong correlation within all papers between discussion of errors and complex cases, and support for error handling. Papers that discuss such cases also tend to implement some mechanism for dealing with errors if they occur. Conversely, papers that do not implement error handling nearly always fail to discuss errors at all.

We identified three popular and recurring categories of error handling mechanisms.

(1) *Overestimation*: For instance, CFG and callgraph overestimation are popular in papers that build binary-level security; it minimizes the risk of accidentally prohibiting valid edges, though the precision of security policies may suffer slightly.

(2) *Underestimation*: This is used in papers where soundness is more important than completeness.

(3) *Runtime augmentation*: Some papers use static analysis to approximate a primitive, and use low-cost runtime checks to fix errors in the primitive where needed.

Overestimation is the most popular error handling strategy, used in around 30% of top-tier papers. It is followed by underestimation and runtime augmentation.

6 Discussion

Our findings show a dualism in the stance on disassembly in the literature. On the one hand, the difficulty of pure (instruction-level) disassembly is often exaggerated. The prevalence of complex constructs like overlapping basic blocks, inline data, and overlapping instructions is frequently overestimated, especially for `gcc` and `clang` [5, 23]. This leads reviewers and researchers to underestimate the effectiveness of binary-based research.

We showed that unless binaries are deliberately obfuscated, instruction recovery is extremely accurate, especially in ELF binaries generated with gcc or clang. We did not find any inline data for these binaries, even in optimized library code; even jump tables are explicitly placed in the .rodata section. Moreover, in Visual Studio binaries with jump tables in the code section, modern disassemblers like IDA Pro recognize and resolve them quite accurately. The rare overlapping instructions in handcrafted library code take on a limited number of forms, typically using a direct conditional jump over a prefix. These are resolved without problems by IDA Pro and Dyninst, among others. The same is true for multi-entry functions, which are also rare. Moreover, overlapping/shared basic blocks (commonly cited as particularly challenging for binary analysis), do not appear in our findings at all.

On the other hand, some primitives really do often suffer from inaccuracies. Some recursive disassemblers used for binary instrumentation (notably Dyninst) regularly miss up to 10% of basic blocks in optimized binaries, calling for special attention in systems which rely on basic block-level binary instrumentation. Additionally, function signatures in 64-bit code are extremely inaccurate; fortunately, they are also rarely used in the literature.

However, function starts *are* regularly needed, though the false negative rate regularly rises to 20% or more even for the best performing disassemblers. This is especially true in optimized binaries, or in coding styles that make extensive use of function pointers. Worse, false positive function starts are almost as common. This can lead to problems in some binary-based research, especially binary instrumentation, if care is not taken to ensure graceful failure in the event of misdetected function starts. Symbols offer a great deal of help, especially in reducing the false negative rate. Unfortunately, they are rarely available in practice.

It is surprising then, to find that only 20% to 25% (top-tier) of Windows papers that use function starts, and 33% to 50% (top-tier) of the Linux papers, devote any attention to discussing these problems. A similarly small number of papers implement error handling, even though errors can cause malignant failures in a majority of papers. While it is not impossible to base well-functioning binary-based systems on function start information (or other primitives), it is crucial that such work implement mechanisms for handling inaccuracies. Three effective classes of error handling (depending on the situation) have already been proposed in the literature: overestimation, underestimation, and runtime augmentation.

We hope our study will facilitate a better match between expectations on disassembly in future research, and the performance actually delivered by modern disassemblers. Moreover, we believe our findings can be used to better judge where problems are to be expected, and to implement effective mechanisms for dealing with them.

7 Related Work

Prior work on disassembly precision focused on complex corner cases [5, 23, 25] or obfuscated code [18, 34], showing that these can strongly reduce disassembly accuracy. We focus instead on the performance of modern disassemblers given realistic full-scale binaries without active anti-disassembly techniques.

Miller et al. center their analysis around complex cases in glibc-2.12 [23]. Their findings largely correspond to our own, though we found no inline jump tables in glibc-2.22. In addition to their glibc analysis, Miller et al. find complex cases in SPEC CPU2006; however, this analysis focuses exclusively on statically linked binaries. We show in Section 3.3 that these cases are entirely due to embedded library code, and are extremely rare in non-statically linked applications.

Our finding that function starts are among the most challenging primitives to recover is in agreement with results by Bao et al. [4].

Paleari et al. study instruction decoders in disassemblers [25], which parse individual x86 instructions. Specific instructions that are sometimes wrongly parsed have also been outlined by the authors of Capstone [31].

Complex constructs in obfuscated code are discussed by Schwarz et al. [34], Linn et al. [21] and Kruegel et al. [18]. We show that these worst-case complex constructs are exceedingly rare in non-obfuscated code.

8 Conclusion

Our study contradicts the widespread belief that complex constructs severely limit the usefulness of binary-based research. Instead, we show that modern disassemblers achieve close to 100% instruction disassembly accuracy for compiler-generated binaries, and that constructs like inline data and overlapping code are very rare. Errors in areas where disassembly is truly lacking, such as function start recovery, are not discussed nearly as often in the literature. By analyzing discrepancies between disassembler capabilities and the literature, our work provides a foundation for guiding future research.

Acknowledgements

We thank the anonymous reviewers for their valuable input to improve the paper. We also thank Mingwei Zhang and Rui Qiao for their proofreading and feedback. This work was supported by the European Commission through project H2020 ICT-32-2014 “SHARCS” under Grant Agreement No. 644571, and by the Netherlands Organisation for Scientific Research through grant NWO CSI-DHS 628.001.021 and the NWO 639.023.309 VICI “Dowsing” project.

References

[1] ANDRIESSE, D., VAN DER VEEN, V., GÖKTAŞ, E., GRAS, B., SAMBUC, L., SLOWINSKA, A., BOS, H., AND GIUFFRIDA, C. Practical Context-Sensitive CFI. In *Proceedings of the 22nd Conference on Computer and Communications Security (CCS'15)* (Denver, CO, USA, October 2015), ACM.

[2] BACKES, M., AND NÜRNBERGER, S. Oxymoron Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Sec'14)* (2014).

[3] BALAKRISHNAN, G., AND REPS, T. WYSINWYX: What You See is Not What You eXecute. *ACM Transactions on Programming Languages and Systems 32*, 6 (Aug. 2010), 23:1–23:84.

[4] BAO, T., BURKET, J., WOO, M., TURNER, R., AND BRUMLEY, D. BYTEWEIGHT: Learning to Recognize Functions in Binary Code. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Sec'14)* (2014).

[5] BERNAT, A. R., AND MILLER, B. P. Anywhere, Any-Time Binary Instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools* (2011).

[6] BONFANTE, G., FERNANDEZ, J., MARION, J.-Y., ROUXEL, B., SABATIER, F., AND THIERRY, A. CoDisasm: Medium Scale Concatic Disassembly of Self-Modifying Binaries with Overlapping Instructions. In *Proceedings of the 22nd Conference on Computer and Communications Security (CCS'15)* (2015).

[7] BRUMLEY, D., JAGER, I., AVGERINOS, T., AND SCHWARTZ, E. J. BAP: A Binary Analysis Platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11)* (2011).

[8] CHEN, X., SLOWINSKA, A., ANDRIESSE, D., BOS, H., AND GIUFFRIDA, C. StackArmor: Comprehensive Protection from Stack-Based Memory Error Vulnerabilities for Binaries. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'15)* (San Diego, CA, USA, February 2015), Internet Society.

[9] CHENG, Y., ZHOU, Z., YU, M., DING, X., AND DENG, R. H. ROPecker: A Generic and Practical Approach for Defending Against ROP Attacks. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'14)* (2014).

[10] DAVI, L., KOEBERL, P., AND SADEGHI, A.-R. Hardware-Assisted Fine-Grained Control-Flow Integrity: Towards Efficient Protection of Embedded Systems Against Software Exploitation. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID'15)* (2015).

[11] EGELE, M., WOO, M., CHAPMAN, P., AND BRUMLEY, D. Blanket Execution: Dynamic Similarity Testing for Program Binaries and Components. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Sec'14)* (2014).

[12] EVANS, I., LONG, F., OTGONBAATAR, U., SHROBE, H., RINARD, M., OKHRAVI, H., AND SIDIROGLOU-DOUSKOS, S. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *Proceedings of the 22nd Conference on Computer and Communications Security (CCS'15)* (Denver, CO, USA, 2015), ACM.

[13] GAWLIK, R., AND HOLZ, T. Towards Automated Integrity Protection of C++ Virtual Function Tables in Binary Programs. In *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC'14)* (2014).

[14] HALLER, I., SLOWINSKA, A., NEUGSCHWANDTNER, M., AND BOS, H. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Sec'13)* (2013).

[15] HU, X., AND SHIN, K. G. DUET: Integration of Dynamic and Static Analyses for Malware Clustering with Cluster Ensembles. In *Proceedings of the 29th Annual Computer Security Applications Conference (ACSAC'13)* (2013).

[16] JANG, J., WOO, M., AND BRUMLEY, D. Towards Automatic Software Lineage Inference. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Sec'13)* (2013).

[17] KINDER, J. *Static Analysis of x86 Executables*. PhD thesis, Technische Universität Darmstadt, 2010.

[18] KRUEGEL, C., ROBERTSON, W., VALEUR, F., AND VIGNA, G. Static Disassembly of Obfuscated Binaries. In *Proceedings of the 13th USENIX Security Symposium (USENIX Sec'04)* (2004).

[19] LAURENZANO, M., TIKIR, M. M., CARRINGTON, L., AND SNAVELY, A. PEBIL: Efficient Static Binary Instrumentation for Linux. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software* (2010).

[20] LEE, K. H., ZHANG, X., AND XU, D. High Accuracy Attack Provenance via Binary-based Execution Partition. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'13)* (2013).

[21] LINN, C., AND DEBRAY, S. Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS'03)* (2003).

[22] MICROSOFT DEVELOPER NETWORK. Overview of x64 Calling Conventions, 2015. `https://msdn.microsoft.com/en-us/library/ms235286.aspx`.

[23] MILLER, B. P., AND MENG, X. Binary Code is Not Easy, 2015. Technical report, University of Wisconsin-Madison.

[24] MOHAN, V., LARSEN, P., BRUNTHALER, S., HAMLEN, K. W., AND FRANZ, M. Opaque Control-Flow Integrity. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'15)* (2015).

[25] PALEARI, R., MARTIGNONI, L., FRESI ROGLIA, G., AND BRUSCHI, D. N-Version Disassembly: Differential Testing of x86 Disassemblers. In *Proceedings of the 19th International Symposium on Software Testing and Analysis* (2010), ISSTA'10.

[26] PENG, F., DENG, Z., ZHANG, X., XU, D., LIN, Z., AND SU, Z. X-Force: Force-Executing Binary Programs for Security Applications. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Sec'14)* (2014).

[27] PEWNY, J., GARMANY, B., GAWLIK, R., ROSSOW, C., AND HOLZ, T. Cross-Architecture Bug Search in Binary Executables. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P'15)* (2015).

[28] PEWNY, J., SCHUSTER, F., ROSSOW, C., BERNHARD, L., AND HOLZ, T. Leveraging Semantic Signatures for Bug Search in Binary Programs. In *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC'14)* (2014).

[29] PRAKASH, A., HU, X., AND YIN, H. vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'15)* (San Diego, CA, USA, February 2015), Internet Society.

[30] QIAO, R., ZHANG, M., AND SEKAR, R. A Principled Approach for ROP Defense. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC'15)* (2015).

[31] QUYNH, N. A. Capstone: Next-Gen Disassembly Framework. In *Blackhat USA* (2014).

[32] ROMER, T., VOELKER, G., LEE, D., WOLMAN, A., WONG, W., LEVY, H., BERSHAD, B., AND CHEN, B. Instrumentation and Optimization of Win32/Intel Executables Using Etch. In *Proceedings of the USENIX Windows NT Workshop (NT'97)* (1997).

[33] SCHWARTZ, E. J., LEE, J., WOO, M., AND BRUMLEY, D. Native x86 Decompilation Using Semantics-Preserving Structural Analysis and Iterative Control-Flow Structuring. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Sec'13)* (2013).

[34] SCHWARZ, B., DEBRAY, S., AND ANDREWS, G. Disassembly of Executable Code Revisited. In *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE'02)* (2002).

[35] SHIN, E. C. R., SONG, D., AND MOAZZEZI, R. Recognizing Functions in Binaries with Neural Networks. In *Proceedings of the 24th USENIX Security Symposium (USENIX Sec'15)* (2015).

[36] SHOSHITAISHVILI, Y., WANG, R., HAUSER, C., KRUEGEL, C., AND VIGNA, G. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware.

[37] TANG, A., SETHUMADHAVAN, S., AND STOLFO, S. Heisenbyte: Thwarting Memory Disclosure Attacks using Destructive Code Reads. In *Proceedings of the 22nd Conference on Computer and Communications Security (CCS'15)* (2015).

[38] TRAIL OF BITS. A Preview of McSema, 2014. Technical report. `http://blog.trailofbits.com/2014/06/23/a-preview-of-mcsema/`.

[39] VAN DER VEEN, V., GÖKTAŞ, E., CONTAG, M., PAWLOSKI, A., CHEN, X., RAWAT, S., BOS, H., HOLZ, T., ATHANASOPOULOS, E., AND GIUFFRIDA, C. A Tough call: Mitigating Advanced Code-Reuse Attacks At The Binary Level. In *Proceedings of the 37th Symposium on Security and Privacy (S&P'16)* (May 2016).

[40] VOGL, S., GAWLIK, R., GARMANY, B., KITTEL, T., PFOH, J., ECKERT, C., AND HOLZ, T. Dynamic Hooks: Hiding Control Flow Changes within Non-Control Data. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Sec'14)* (2014).

[41] WANG, M., YIN, H., BHASKAR, A. V., SU, P., AND FENG, D. Binary Code Continent: Finer-Grained Control Flow Integrity for Stripped Binaries. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC'15)* (2015).

[42] WANG, S., WANG, P., AND WU, D. Reassembleable Disassembling. In *Proceedings of the 24th USENIX Security Symposium (USENIX Sec'15)* (2015).

[43] WARTELL, R., ZHOU, Y., HAMLEN, K. W., KANTARCIOGLU, M., AND THURAISINGHAM, B. M. Differentiating Code from Data in x86 Binaries. In *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases* (2011).

[44] YAKDAN, K., ESCHWEILER, S., GERHARDS-PADILLA, E., AND SMITH, M. No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantics-Preserving Transformations. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'15)* (2015).

[45] ZHANG, C., SONG, C., CHEN, K. Z., CHEN, Z., AND SONG, D. VTint: Protecting Virtual Function Tables' Integrity. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'15)* (2015).

[46] ZHANG, C., WEI, T., CHEN, Z., DUAN, L., SZEKERES, L., MCCAMANT, S., SONG, D., AND ZOU, W. Practical Control Flow Integrity and Randomization for Binary Executables. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P'13)* (2013).

[47] ZHANG, M., QIAO, R., HASABNIS, N., AND SEKAR, R. A Platform for Secure Static Binary Instrumentation. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'14)* (2014).

[48] ZHANG, M., AND SEKAR, R. Control Flow Integrity for COTS Binaries. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Sec'13)* (2013).

[49] ZHANG, M., AND SEKAR, R. Control Flow and Code Integrity for COTS binaries. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC'15)* (2015).

[50] ZHANG, Y., SUN, X., DENG, Y., CHENG, L., ZENG, S., FU, Y., AND FENG, D. Improving Accuracy of Static Integer Overflow Detection in Binary. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID'15)* (2015).