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

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

[†]Vrije Universiteit Amsterdam
[§]Lastline, Inc.

USENIX Security 2016

Introduction

Disassembly in Systems Security

Disassembly is the backbone of all binary-level systems security work (and more)

- Control-Flow Integrity
- Automatic Vulnerability/Bug Search
- Lifting binaries to LLVM/IR (e.g., for reoptimization)
- Malware Analysis
- Binary Hardening
- Binary Instrumentation
- . . .

Introduction

Challenges in Disassembly

Disassembly is undecidable, and disassemblers face many challenges

- Code interleaved with data
- Overlapping basic blocks
- Overlapping instructions (on variable-length ISAs)
- Indirect jumps/calls
- Alignment/padding bytes (such as `nops`)
- Multi-entry functions
- Tailcalls
- ...

How much of a problem do these challenges cause in practice?

Introduction

Motivation of our Work

Prior work explores corner cases, but no consensus on how common these really are in practice

- Pessimistic view of disassembly among reviewers and researchers
- Underestimation of the potential of binary-based work

We study the frequency of corner cases in real-world binaries, and measure how well disassemblers deal with them

Experiment Setup

Binary Types

We cover a wide range of commonly targeted binary types (*981 tests*)

- SPEC CPU2006 + real-world applications (C and C++)
- Compiled with `gcc`, `clang` (ELF) and Visual Studio (PE)
- Compiled for x86 and x64
- Five optimization levels (`O0`-`O3` and `Os`) + `-flto`
- Dynamically and statically linked binaries
- Stripped binaries and binaries with symbols
- Library code with handwritten assembly (`glibc`)

Focus on benign use cases, such as binary protection schemes (we already know obfuscated binaries can wreak havoc)

Experiment Setup

Ground Truth

Ground truth from DWARF/PDB, with source-level LLVM info

Disassembly Primitives and Complex Cases

We study five commonly used disassembly/binary analysis primitives

- ① Instructions, ② Function starts, ③ Function signatures, ④ Control Flow Graph (CFG) accuracy, ⑤ Callgraph accuracy

Measure prevalence of seven complex cases

- ① Overlapping BBs, ② Overlapping instructions, ③ Inline data/jump tables, ④ Switches, ⑤ Padding bytes, ⑥ Multi-entry functions, ⑦ Tailcalls

Disassemblers

Tested nine popular industry and research disassemblers (details in paper and in results where needed)

Experiment Results

More results

Far too many results to fit in this presentation

- Focus on most interesting results here, see paper for more
- **Detailed results and ground truth publicly released**
  `https://www.vusec.net/projects/disassembly/`

Experiment Results

Instruction Accuracy

Very high accuracy for best performing disassemblers

- IDA Pro 6.7: 96%–99% TP (FNs due to padding, FPs rare)
- Linear: **100% correct on ELF (no inline data)**
  99% correct for PE, some FPs/FNs due to inline jump tables

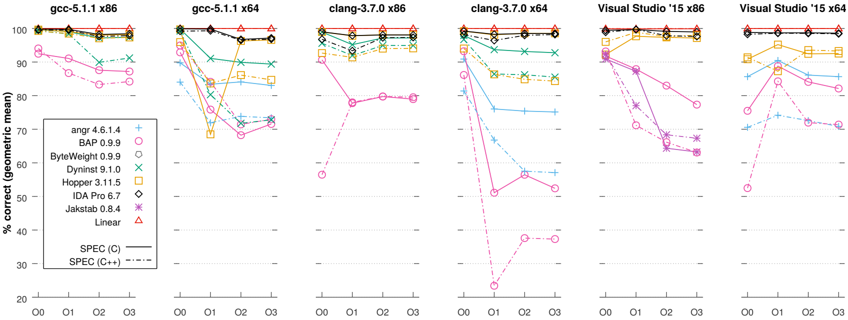

Figure: Correctly disassembled instructions

Experiment Results

CFG and Callgraph accuracy

CFG and callgraph very accurate due to high instruction accuracy (see paper for details)

Experiment Results

Function Signatures

Only IDA Pro, important mostly for manual reverse engineering

- Poor accuracy, especially on x64
- Acceptable for manual analysis, caution in automated analysis

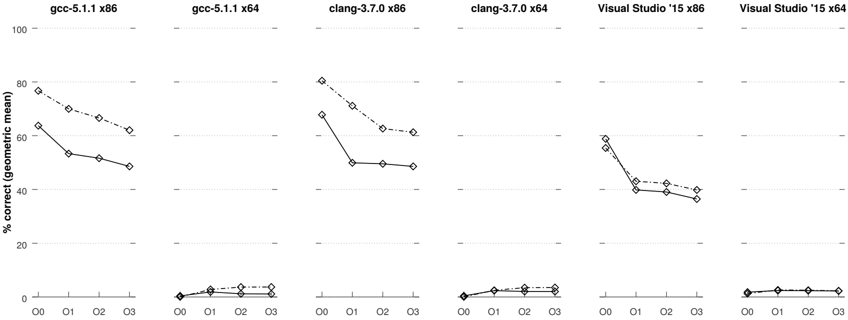

Figure: Correctly detected non-empty argument list (IDA Pro, `argc` only)

Experiment Results

Function Detection

Function detection currently the main disassembly challenge

- Even function start detection yields many FPs/FNs (20%+)
- Complex cases: non-standard prologues, tailcalls, inlining, ...
- Binary analysis commonly requires function information

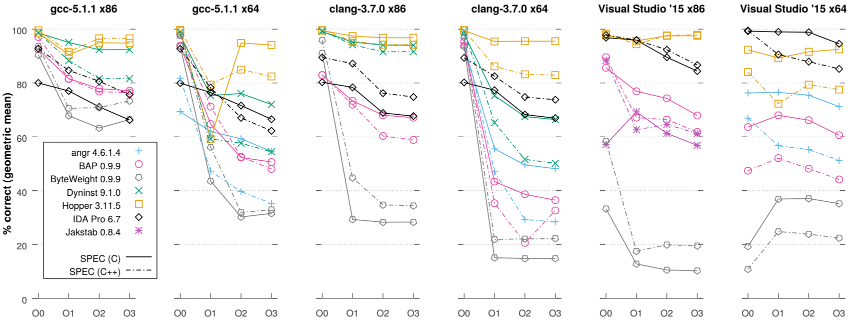

Figure: Correctly detected function start addresses

Experiment Results

Function Detection: False Negative

Listing: False negative indirectly called function for IDA Pro 6.7 (gcc compiled with gcc at O3 for x64 ELF)

```
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
```

Experiment Results

Function Detection: False Positive

Listing: False positive function (shaded) for Dyninst (`perlbench` compiled with `gcc` at `O3` for x64 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
```

Prevalence of Complex Cases

Complex Cases in Application Code

- **No inline data in ELF**, even jump tables placed in `.rodata`
- Inline data for PE (jump tables), well recognized by IDA Pro
- **No overlapping basic blocks**, contrary to widespread belief
- **Tailcalls quite common** (impact on function detection)

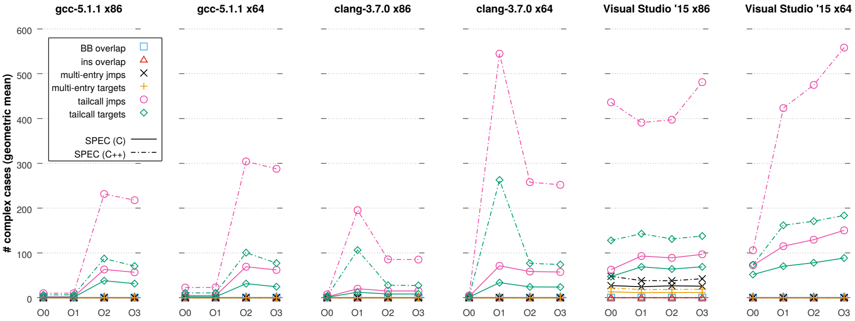

Figure: Prevalence of complex constructs in SPEC CPU2006 binaries

Prevalence of Complex Cases

Complex Cases in Library Code (`glibc-2.22`)

Highly optimized library code (handwritten assembly) allows for more complex cases

- Surprisingly, **no inline data** in recent `glibc` versions (explicitly pushed into `.rodata` even in handwritten code)
- **No overlapping basic blocks**
- **Tailcalls again quite common**
- **Some overlapping instructions** (handwritten assembly)
- **Some multi-entry functions** (well-defined)

Prevalence of Complex Cases

Complex Cases in Library Code: Overlapping Instruction

Listing: Overlapping instruction in `glibc-2.22`

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

Prevalence of Complex Cases

Complex Cases in Library Code: Multi-Entry Function

Listing: Multi-entry function 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>
  [...]
```

Disassembly in the Literature

Comparison of Results

Compared our results to the requirements and expectations of disassembly-based security work published between 2013–2015

- Instructions/CFG information needed in nearly all papers
- Function detection required by half of the papers
- Linear disassembly rarely used, even when more accurate (ELF)
- **Only 30% of papers that use function detection discuss potential errors**, despite its unreliability
- **Errors in function detection are discussed less often than for any other primitive**
- **In 70% of papers, errors are fatal** (unusable results or crashes)
- **Only 43% of papers handle errors in any primitive**
- Most papers that handle errors use overestimation (conservative analysis) or runtime fixes

Discussion and Conclusion

Expectations of disassembly are mismatched with actual results

- Research focuses on extremely rare or nonexistent corner cases
- Function detection currently biggest challenge, but errors discussed more rarely than any other primitive
- Few papers implement mechanisms for handling disassembly errors, even when these are fatal

Real-world data on disassembly enables better judgement of directions for future work

- Many more results and details given in our paper
- **Detailed results and ground truth publicly released** `https://www.vusec.net/projects/disassembly/`