

THE INCONVENIENT TRUTHS OF GROUND TRUTH FOR BINARY ANALYSIS

JIM ALVES-FOSS AND VARSHA VENUGOPAL



INTRODUCTION

The effectiveness of binary analysis tools and techniques is often measured with respect to how well they map to a *ground truth*.

We have found that not all ground truths are created equal.

This paper challenges the binary analysis community to take a long look at the concept of ground truth, to ensure that we are in agreement with definition(s) of ground truth, so that we can be confident in the evaluation of tools and techniques.

This becomes even more important as we move to trained machine learning models, which are only as useful as the validity of the ground truth in the training.



DEFINING GROUND TRUTH

The definition of ground truth, with respect to a binary executable, is context dependent.

- A lot depends on what the researchers are examining: correct decoding of instructions and data, mapping of binary to complex structures such as functions or data structures, or mapping of binary back to the source code.
- This context matters when evaluating the effectiveness of binary analysis tools.
- It also matters if we are examining normal code, obfuscated code or malware. For the purposes of this paper, we will primarily focus on normal code.



INSTRUCTIONS

Where do we obtain the ground truth about which bytes are instructions?

- Do we need to instrument the compiler, or is their sufficient details in the debug data?

What about instructions as data?

- There are some programs that run checksums over their own code, in this case code is data -- most people will define that code as instructions. However, other programs may store portions of code that they copy to other parts of memory. This may be malicious, or may be part of a just-in-time compilation routine. Is that stored code really instructions, or just data?



INSTRUCTIONS (2)

Recommendation: We recommend defining instructions as executable code as compiled into the binary. Instructions that are only meant to be copied, and are therefore used only as data, should be categorized based on location. If these are stored in the data section, then they are data. If these instructions are stored in the code section, they are code, even if they are never called or executed



FUNCTIONS (1)

Is there a need to map binary directly to source code?

- If so, what happens when a function is in-lined?
- What happens if the compiler creates multiple entry points to the function, or divides the function into two separate functions?

What is a function within the binary?

- Do we define a function with a strict adherence to the source code?
- What about a function with a tail call (last instruction is a function call)?
- If the compiler optimizes away the call with a jump, are we jumping to a new function or to a disjoint portion of the same function?
- What about a compiler that allows a function to fall through to the next function?



FUNCTIONS (2)

Are uncalled functions still functions?

- Some compilers may optimize them away, while others do not.

Are compiler added functions, functions?}

- There are several functions added by compilers, many are hand coded assembly.

Can functions have more than one entry point?

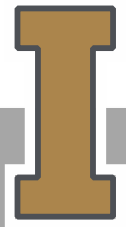
- Some compilers will optimize a function into multiple separate functions.
- Also, some compilers may insert multiple entry points for same function. The Intel compiler does this in 32-bit



FUNCTIONS (3)

Besides function entry points, how do we define function boundaries?

- Some tools look for the start and end bytes of a function.
- Is the listed end byte the last byte of the last instruction? The first byte of the last instructions? The first byte after the last instruction?
- And how about padding between functions, does that belong to a specific function.



FUNCTIONS (4)

Recommendation: We recommend defining functions as logical units within the binary that have an entry point and one or more exit points and the collection of executable instructions in between (including exception handlers). The exit may be a call to a non-returning function, a jump to the start of another function, or a return. This maps as close as we can to the source code.

- We do not count in-line functions as functions, but rather treat them as if they were macros.
- We do count compiler inserted functions as functions, since they are in the binary.



CROSS REFERENCES/POINTERS, INDIRECT JUMPS AND CALLS (1)

Within the code there will be pointers/references to other code and data.

Within the data section, there will be pointers/references to code and data.

These pointers/references may be absolute values, or relative. We don't think there are too many questions related to these values as they are what they are.

There are jump tables, function pointers and other values that are used for indirect jumps and calls.

The location of these values fits into this category as pointers/references to code.



CROSS REFERENCES/POINTERS, INDIRECT JUMPS AND CALLS (2)

Where do we obtain the ground truth about which bytes are these references/pointers?

Some people recommend using relocation data for pointers, but that does not work well in position independent code which does not need as much relocation information. Others will instrument the compiler.

Recommendation: These values are what they are. We don't think instrumenting a compiler to find these values is the best long-term solution for generating the ground truth. If debug data contains all typing information, we can extract it from there.



SPECIAL FUNCTIONS (1)

There are functions that do not return, and these affect the validity of control flow analysis. There are also functions that are inserted into the binary by the compiler, and are not linked to the source code.

For non-returning functions, where do we get the ground truth about this characteristic?

- We have seen tools that embed a list of non-returning standard library functions. This need to be updated for the libraries.

How does a non-returning function affect control flow?

- Is an instruction after a call to a non-returning function part of the same function or not? We have seen fault-tolerant code that add additional instructions after calls to non-returning functions, just in case an error results in a return. The ground truth of the program must capture this.



SPECIAL FUNCTIONS (2)

Recommendation:

Non-returning functions are special and need to be recognized and documented.

- OS system calls may need to be documented with the non-returning feature.
- Everything else should be recursively analyzable. Difficulties may arise with function pointers and virtual functions.

Any functions inserted by the compiler are just part of the program and need to be treated as such.



EXAMPLES (1)

Multiple entry points for a binary (icc compiler)

```
fix_syms(): ../binutils2.23/bfd/linker.c:3208
```

```
080b41c0 <fix_syms>:
```

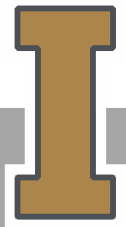
```
80b41c0:  mov    0x4(%esp),%eax
```

```
80b41c4:  mov    0x8(%esp),%edx
```

```
080b41c8 <fix_syms.>:
```

```
80b41c8:  push  %esi
```

```
80b41c9:  push  %edi
```



EXAMPLES (2)

Symbol table from `as_new`, compiled with `icc` (partial clones)

08055750	1	F	.text	00000c30	operand..0
080570a0	1	F	.text	00000330	integer_constant..0
08056b60	1	F	.text	00000540	integer_constant..2
08056840	1	F	.text	00000320	integer_constant..3
08056520	1	F	.text	00000320	integer_constant..4
0805a7d0	1	F	.text	00000cb0	expr..0
080573d0	1	F	.text	00000c80	expr..1
08058fa0	1	F	.text	00000cd0	operand
08056380	1	F	.text	000001a0	integer_constant..1



EXAMPLES (3)

integer_constant..2 (radix = 16)

/data/usenix/Linux/binutils-2.23/gas/expr.c:360

number = number * radix + digit;

```
8056baa: 8b d5          mov    %ebp,%edx
8056bac: c1 ea 1c       shr    $0x1c,%edx
8056baf: c1 e5 04       shl    $0x4,%ebp
8056bb2: c1 e6 04       shl    $0x4,%esi
```

integer_constant..3 (radix = 2)

/data/usenix/Linux/binutils-2.23/gas/expr.c:360

number = number * radix + digit;

```
805688c: 8b d3          mov    %ebx,%edx
805688e: 03 db          add    %ebx,%ebx
8056890: c1 ea 1f       shr    $0x1f,%edx
8056893: 03 c0          add    %eax,%eax
```



EXAMPLES (4) –

```
int main (int argc, char **argv)
{
    ...
    for (size_t copies = bufalloc / copysize; --copies; )
    {
        memcpy (buf + bufused, buf, copysize);
        bufused += copysize;
    }

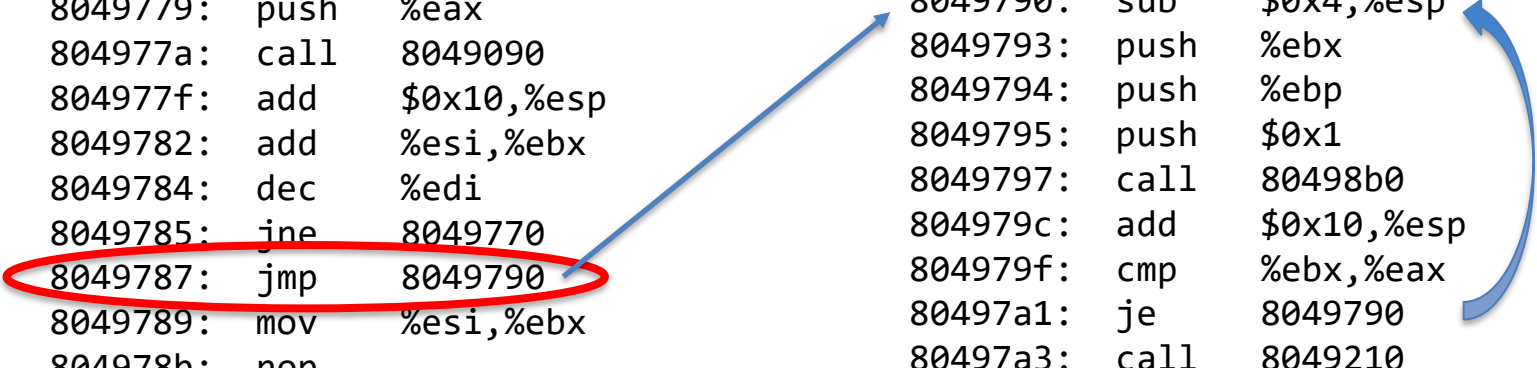
    /* Repeatedly output the buffer until there is a write error; then fail. */
    while (full_write (STDOUT_FILENO, buf, bufused) == bufused)
        continue;
    error (0, errno, _("standard output"));
    return EXIT_FAILURE;
}
```



EXAMPLES (4)

```
8049770: lea    0x0(%ebp,%ebx,1),%eax    . . .
8049774: sub    $0x4,%esp
8049777: push  %esi
8049778: push  %ebp
8049779: push  %eax
804977a: call  8049090
804977f: add    $0x10,%esp
8049782: add    %esi,%ebx
8049784: dec    %edi
8049785: jne    8049770
8049787: jmp    8049790
8049789: mov    %esi,%ebx
804978b: nop
804978c: nop
804978d: nop

804978e: nop
804978f: nop
8049790: sub    $0x4,%esp
8049793: push  %ebx
8049794: push  %ebp
8049795: push  $0x1
8049797: call  80498b0
804979c: add    $0x10,%esp
804979f: cmp    %ebx,%eax
80497a1: je     8049790
80497a3: call  8049210
```



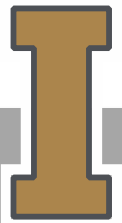
IDA PRO thinks this is last address of function



EXAMPLES (5)

```
4056b5:  add    %rax,%rdx
4056b8:  mov    (%rdx),%rax
4056bb:  jmpq   *%rax
4056bd:  movzbl -0x120(%rbp),%eax
4056c4:  movzbl %a1,%eax
4056c7:  test   %eax,%eax
4056c9:  je     40580f
. . .
```

IDA PRO can not determine indirect jump addresses, so assumes this is the last instruction of the function.



DWARF

We have seen several papers state that they use debug data for ground truth, but never elaborate further.

When looking at Dwarf debug data, we have seen several issues.

- First involves correctly interpreting the data. The HIGH PC value in a subprogram is the byte after the end of the function. However, it may be an absolute value or a relative value (length), which is not parsed correctly by some libraries/tools.
- Second, not all compilers include complete information in the DWARF data.



DWARF data is a good source, but researchers have to be careful.

COMPILER HACKS

There are a couple of tools that generate ground truth by hacking the compiler.

- Pang et al. where the authors revise the compiler to emit all of the ground truth information they need.
- Li et al. use intermediate representation, such as generated assembly code listings, to assist in the generation of their ground truth for disassembles.

These techniques only work for the compilers they are designed for, and therefore can not be reliably used for generalization of ground truth, even with newer versions of the same compilers.



CONCLUSION (1)

Knowing the ground truth is essential when evaluating the effectiveness of binary analysis tools. We have seen a few instances where the ground truth was incomplete, misleading, misinterpreted or even hacked to get results that the authors wanted. We are not saying that the authors deliberately misled the community, but rather did not focus on the importance of making sure the ground truth was correct. Most authors do not communicate the details of their generation of ground truth or the assumptions they made when doing the evaluation.



CONCLUSION (2)

Without the existence of well vetted tools and/or data sets for ground truth, we will struggle with the ability to accurately build, evaluate and gauge binary analysis tools. If researchers then use incorrect ground truth when using machine learning or other automated analysis, the problem will just get worse.

We recommend a discussion among the community about the types of ground truth metrics we need, the best ways to develop them, and a process for vetting and sharing ground truth generation tools.

We do not believe custom tools, such as compilers modifications, as a good long term solution to ground truth generation.



Use of DWARF debug data and the compiler generation symbol tables is a good start, but their limits need to be fully explored.

CONCLUSION (3)

It would be really useful to have a fully vetted library of binaries, complete with source code, full debug data and database containing all of the ground truth data we are interested in – for a wide variety of microprocessor architectures, operating systems, compilers, and programming languages.

