The Challenge of Indirection: Treating Flags During Sound Analysis of Machine Code

Sven Mattsen, Sibylle Schupp
Hamburg University of Technology, Hamburg, Germany
{sven.mattsen, schupp}@tuhh.de

1 Introduction

A key problem in reverse engineering executables is to reconstruct the programs's control flow, i.e., to construct a graph representation of the byte sequences or paths a computer may execute as instructions. Unfortunately, exactly computing all feasible paths of a program is not possible. Therefore, the task of precise control flow reconstruction and disassembly is also not solvable in general, and one must be content with an approximative answer.

Contemporary disassemblers, such as `objdump`, `radare2`, and `IDAPro` either use linear sweep[5] or recursive descent[5] techniques, enhanced by heuristics, to compute an approximated disassembly. As a result, they may produce assembly that not only omits paths that were feasible in the analyzed binary, but also contain paths that were infeasible in the analyzed binary. While this form of approximation is acceptable for certain tasks, e.g., program understanding, it is unacceptable for tasks such as verification, where one reasons about statements that must hold on all paths (e.g., there exists no path containing an exploitable sequence).

One approach to enable tasks such as verification is to allow only overapproximation, i.e., the reconstructed control flow must contain at least all paths that are feasible in the analyzed executable, but may contain additional paths. Tools that overapproximate in this way are called sound. When a verification task requires proving that an illegal program state cannot be reached, a sound disassembly can be used. If no illegal program state can be reached in the sound disassembly, no illegal program state is reachable in the original program. If it cannot be shown that no illegal program state can be reached in the sound disassembly, then it is unclear if a fault exists or not.

In the following sections, we will first explain sound control flow reconstruction via value set analysis[1] (Section 2) and introduce a challenge that is unique to the analysis of machine code (Section 3). In sections 3.2 and 3.3, we will present two approaches to this problem that we implemented in our BDDStab[4] plug-in for the software analysis framework Jakstab[3].

2 Sound Analysis of Machine Code

The most challenging aspect of sound control flow reconstruction is to compute a precise set of possible addresses for dynamic jumps, i.e., jumps that use a nonconstant target (`jmp %eax`). In BDDStab, we use abstract interpretation techniques, which are commonly used on structured programs, to statically compute an overapproximated set of possible integer values for each register and memory location. The technique requires a memory-efficient set data structure for integers as well as precise algorithms that model the semantics of machine instructions on our overapproximated sets. We use binary decision diagrams (BDD) to enable size efficent storage of nearly full sets of 64-bit integers and define our algorithms on the structure of the BDDs rather than the set they represent, which allows efficient operations for large sets.

3 Flags in Executables

Besides precise models for bitwise and arithmetic instructions, computing precise sets of possible values also requires models for conditional branches, where the abstract states must be restricted according to the condition. These models have to account for the different ways in which conditional branches operate in machine code, namely the indirection via flags. Consider the following example C code and its corresponding assembly:

```
if((unsigned int) argc < 10)
  return 1; else return 0;
```

```
        cmpl  $0x9,-0x4(%rbp)
        ja    $RET1
        mov   $0x1,%eax
        jmp   $RET2
RET1:   mov   $0x0,%eax
RET2:   retq
```

The assembly code first uses `cmpl` to set various flags that are subsequently used in the `jg` operation, which jumps depending on the flag values. This behaviour is different from the C program in that the C program directly refers to the variable argc in the condition. In the assembly version, however, when looking at the jg alone allows only the restriction of flag values, not the value at -0x4(%rbp) which is argc. Because some compilers may interleave other instructions between cmpl and jg, especially on architectures where pipelining is heavily influencing the optimal order of instructions (MIPS), these two instructions cannot easily be treated as a unitary instruction. An analyzer must therefore find a way to restrict the objects of conditions even though they are not explicitly mentioned in the condition of the jump itself.

3.1 Forward Substitution

Our treatment of flags during analysis uses forward substitution as implemented in the binary analysis framework Jakstab[3]. Instead of analyzing on the level of assembly, with Jakstab, analyses are defined on an intermediate language that makes all effects of instructions explicit. Line 1 (cmpl) and 2 (ja) will be translated to the following, where PC is the program counter:

```
1  CF := ($0x9 <u -0x4(%rbp))
2  ZF := ($0x9 = -0x4(%rbp))
3  PC := $RET1 if ~(%CF | %ZF)
```

Forward substitution would replace %CF and %ZF with the right hand side expression of lines 1 and 2 in the example. The analyzer's task is to restrict incoming sets of possible values for the heap value -0x4(%rbp) according to the resulting formula.

3.2 Pattern-based Back Translation

The key observation behind pattern-based backtranslation is that the complex formulas that result from forward substitution are often the result of a compilation of much simpler, semantically equivalent formulas. In our last example, the conditional jump was caused by a simple unsigned less than, and it should be possible to translate the result of forward substitution back to that. To that end, we refined the simplification rules in Jakstab, so that they mirror the patterns for intermediate code generation.

3.3 Constraint Solver

The pattern-based back translation approach does not always work, e.g., when more complex conditions are used in the source language or when assembly is written by hand. For these cases, we have developed a simple, overappxomating constraint solver that, given a boolean formula over base constraints such as *less than*, *greater than* and *equality* predicates, produces overapproximated sets of possible values for the registers and memory locations refered to in the base constraints. This constraint solver uses the fast algorithms for *and*, *or*, and *negation*, available for BDDs. The constraint solver recurses on the AST of the constraint until it reaches a base constraint, generates a set of possible values for all syntactically mentioned registers and memory locations, and propagates these sets upwards. Subsequently, these upwards propagated sets are intersected for boolean *and* nodes, unioned for boolean *or* nodes and complemented for boolean *negation*. Because the complementation of an overapproximated set yields an underapproximation, the constraint solver additionally tracks whether an overapproximation or an underapproximation is needed for each base constraint, whereby it ensures to only generate overapproximations at the top AST node.

4 Related Work

At its core, the indirection problem of flags in binary analysis is a problem of relations between flags and registers and memory locations. However, the shape of the relations is not supported by traditional relational analyses such as affine relations[2]. Sepp et al. [6] therefore introduce virtual flags and corresponding virtual instructions that produce relations that a subsequent affine relations analysis supports.

5 Conclusion

We have implemented the pattern-based and constraint solver approaches in our BDDStab plug-in and found that the pattern-based back translation approach improves precision in the most frequent cases and additionally simplifies the work for the constraint solver if simplification does not yield an expression without boolean connectives.

References

[1] G. Balakrishnan and T. Reps. “Analyzing Memory Accesses in x86 Executables”. In: *Compiler Construction.* LNCS. 2004, pp. 5–23.

[2] M. Karr. “Affine relationships among variables of a program”. In: *Acta Informatica* (1976), pp. 133–151.

[3] J. Kinder and H. Veith. “Jakstab: A Static Analysis Platform for Binaries”. In: *Computer Aided Verification.* LNCS. 2008, pp. 423–427.

[4] S. Mattsen, A. Wichmann, and S. Schupp. “A non-convex abstract domain for the value analysis of binaries”. In: *International Conference on Software Analysis, Evolution and Reengineering.* SANER. 2015, pp. 271–280.

[5] B. Schwarz, S. Debray, and G. Andrews. “Disassembly of Executable Code Revisited”. In: *Working Conference on Reverse Engineering.* WCRE. 2002, pp. 45–.

[6] A. Sepp, B. Mihaila, and A. Simon. “Precise Static Analysis of Binaries by Extracting Relational Information”. In: *Working Conference on Reverse Engineering.* WCRE. 2011, pp. 357–366.