BDDStab: BDD-based Value Analysis of Binaries

Sven Mattsen
Hamburg University of Technology
sven.mattsen@tuhh.de

Arne Wichmann
Hamburg University of Technology
arne.wichmann@tuhh.de

Sibylle Schupp
Hamburg University of Technology
schupp@tuhh.de

1. Introduction

Value analyses compute for each variable a superset of possible values, called variation domain (VD). The results of value analyses are helpful for verification as well as program comprehension and are often used to enable or improve other analyses. For high-level languages like C, value analyses are utilized, for example, to compute targets of dynamic call sites.

For executables, value analyses could be similarly useful, for example, for control-flow reconstruction. Compared to value analyses on high-level languages, the analysis of executables poses the following challenges:

1. Variables may be used as targets in indirect jumps, demanding precise representation of the target's VD.
2. Values are more likely to be the result of both bitwise and arithmetic operations, hence both operations must be computed precisely, even if mixed.
3. Conditional jumps are predicated on formulas that may contain conjunctions, disjunctions, and negation, requiring a suitable constraint solver.

Most value analysis tools require VDs to have a convex shape, that is, all values on a line between any two included values must be included as well. This requirement allows many operations to be executed on the borders of VDs only, which makes such domains efficient, even for large operands. On the down side, VDs of convex domains cannot have holes. For domains that track VDs for each variable independently, convexity implies that if a and b are included in a VD A, then $\{x \mid a \leq x \leq b\} \subseteq A$. For executables, convexity is problematic because targets of indirect jumps are often loaded from tables that contain addresses of code, and such addresses are essentially random; compilers often create such jump tables for `switch` statements. Further, because of compiler optimizations, executables are more likely to contain bitwise operations, which generally do not produce convex sets. We conclude, therefore, that convex domains are not suited for the analysis of binary executables.

Naively implemented, non-convex sets for integer types are inefficient, because storing every element explicitly may require too much space. Further, in the absence of information about relations, binary abstract operations $A \mathbin{\widehat{\circ}} B$ will have to consider each combination of elements in A and B, to produce the required $\{a \circ b \mid a \in A, b \in B\}$, making the operation too inefficient for large operands.

The following well known approaches mitigate these shortcomings:

- Limit set size; introduce explicit $\top$ element (e.g. k-sets)
- Limited disjunctive refinements of convex domains (e.g. k-sets of convex abstract elements, e.g. [3])
- Product of specialized property spaces; concretization uses meet over individual concretizations (Direct Product[2])

Sadly, these methods, applied to contemporary abstract domains, e.g., a direct product of a k-set domain and a disjunctive refinement of an interval domain[3], cannot precisely represent or compute a mix of arithmetic and bitwise operations.

2. BDDStab

BDDStab is a plug-in for the binary analysis framework Jakstab[6]. In difference to the standard interval and k-set abstract domains of Jakstab, BDDStab allows an unlimited number of disjunctions, i.e., sets of arbitrary size, by basing the integer set representation on binary decision diagrams (BDDs) [1]. Any integer can be converted to a sequence of booleans using the two's complement. We use this fact to store n-bit integer sets X_n in a BDD by letting it represent the indicator function $i^{X_n}(x) : \{\mathtt{1}, \mathtt{0}\}^n \rightarrow \{\mathtt{1}, \mathtt{0}\}$ that returns $\mathtt{1}$ exactly when $x \in X_n$. Bits are ordered from most significant (MSB) to least significant bit (LSB) because that allows us to find the largest and smallest elements of a set quickly and enables efficient conversion from and to intervals, which is the fallback for operations without BDD-specific algorithms. BDDs not only make the representation size-efficient for non-convex sets, but also support efficient union and intersection. Additionally, we introduce specialized algorithms for bitwise operations, subtraction, and addition that operate directly on the BDD's structure. Thereby, these algorithms remain efficient, even for large operands. Note that we use complemented edges as decribed by Madre and Billon[7] , which further optimizes the BDD size and allows $O(1)$ complementation of sets. The fast complementation is helpful for our implementation of an extensible, overapproximating constraint satisfaction solver that we use in combination with forward substitution[4, 8] to restrict VDs at conditional branches.

3. Case Study

As an example, let us discuss the `demo2.exe` example, taken from Kinder's paper about alternating control-flow reconstruction [5]. The `demo2.exe` binary contains code sequences, identified as challenging in Section 1. The corresponding assembly code, with added edges to visualize control flow originating from jump instructions and shortened addresses for readability, is depicted in Figure 1. Note that the edges represent exactly the jump information that

TAPAS '14, September 10, 2014, Munich, Bavaria, Germany.
Copyright is held by the owner/author(s).

Jakstab[6], using our domain, will determine. The control flow of the assembly code can be divided into two parts; a loop part from 0x00 to 0x33 that is terminated by a counter; and a loop-free part from 0x35 to 0x58. Looking only at the code, without information about possible jump targets, it is not obvious that the jump at 0x41 will not cause a loop. The method in the original paper found that only 35 of the 37 instructions are reachable that `objdump -d`[1] disassembles. Although not specified in the paper, the code that was found dead is likely at the addresses 0x02 and 0x03, as indicated in the original assembly. Using our BDD-based domain to resolve the targets of jumps, Jakstab is able to additionally determine that the code at the addresses 0x4a and 0x4c is dead. There are three crucial facts our domain must find to identify the dead code:

1. The VD of %ebx for the indirect jump at 0x33 must not contain the addresses of dead code. Note that in practice, any approximation of VDs for indirect jumps causes approximation at subsequent program locations, which may then cause dead code to be identified as reachable.
2. Similarly, the VD of %eax for the indirect jump at 0x41 may not be approximated.
3. It must be shown that %eax and %ebx cannot be equal at 0x3d, to show that the jump at 0x3f will not be taken.

In the following in-depth discussion, we will take a look at the contents of the %eax and %ebx registers as well as that of the memory location 0x00403000, which stores a counter value that is 0 initially.

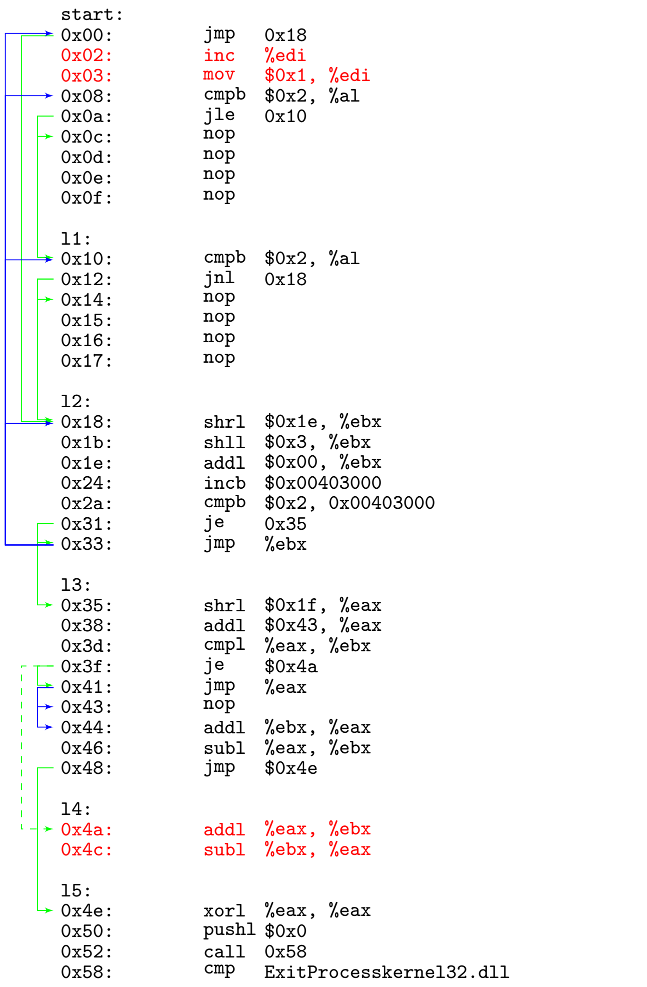
Table 1 lists a selection of flow data after stabilization as determined by our BDD-based abstract domain. We denote the variation domains of %eax and %ebx in bits, where * indicates that the value of a bit is not known. Because nothing is known about %eax and %ebx initially, both are set to $\top$ in 0x00 and no subsequent join operation will improve precision. After the shift instruction at 0x18, which shifts 30 times to the right and fills with zeros from the left, only the least significant two bits of %ebx remain unknown. The subsequent shift instruction will shift these bits to the left by 3 positions, and the following `addl` instruction does not interfere with the unknown bits. The code from 0x24 to 0x31 increments the counter, which terminates the loop that results from the jump instruction at 0x33 after the second iteration. A look at the flow data shows that the indirect jump at 0x33 can only have the targets 0x00, 0x08, 0x10, and 0x18 (Fact 1). When control reaches 0x35, %eax gets shifted by 31 positions to the right and filled with zeros from the left. After the shift, all but the least significant bit of %eax are known to be 0. As a result of the addition in 0x38, our domain determines that %eax can only have the values 0x43 and 0x44, thereby generating precise jump targets for the jump at 0x41 (Fact 2). Because the intersection between the variation domains of %eax and %ebx at 0x3d is empty, the corresponding jump at 0x3f cannot be taken (Fact 3). Therefore, our domain can determine that the code at 0x4a and 0x4c is dead as well.

We provide BDDStab as well as other material needed to reproduce the case study at `http://www.sts.tu-harburg.de/research/bddstab.html`.

[1] `objdump` uses a simple linear sweep disassembler.

Location	Counter	Registers	
0x00	$\{0,1\}$	%eax	0b **** **** **** **** **** **** **** ****
		%ebx	0b **** **** **** **** **** **** **** ****
0x08	$\{0,1\}$	%eax	0b **** **** **** **** **** **** **** ****
		%ebx	0b **** **** **** **** **** **** **** ****
0x10	$\{0,1\}$	%eax	0b **** **** **** **** **** **** **** ****
		%ebx	0b **** **** **** **** **** **** **** ****
0x18	$\{0,1\}$	%eax	0b **** **** **** **** **** **** **** ****
		%ebx	0b 0000 0000 0000 0000 0000 0000 0000 00**
0x1b	$\{0,1\}$	%eax	0b **** **** **** **** **** **** **** ****
		%ebx	0b 0000 0000 0000 0000 0000 0000 000* *000
0x1e	$\{0,1\}$	%eax	0b **** **** **** **** **** **** **** ****
		%ebx	0b 0000 0000 0100 0000 0001 0000 000* *000
0x24	$\{1,2\}$	%eax	0b **** **** **** **** **** **** **** ****
		%ebx	0b 0000 0000 0100 0000 0001 0000 000* *000
0x23	$\{1,2\}$	%eax	0b **** **** **** **** **** **** **** ****
		%ebx	0b 0000 0000 0100 0000 0001 0000 000* *000
0x33	$\{1,2\}$	%eax	0b **** **** **** **** **** **** **** ****
		%ebx	0b 0000 0000 0100 0000 0001 0000 000* *000
0x35	$\{2\}$	%eax	0b 0000 0000 0000 0000 0000 0000 0000 000*
		%ebx	0b 0000 0000 0100 0000 0001 0000 000* *000
0x38	$\{2\}$	%eax	0b 0000 0000 0100 0000 0001 0000 0100 0011
			0b 0000 0000 0100 0000 0001 0000 0100 0100
		%ebx	0b 0000 0000 0100 0000 0001 0000 000* *000
0x41	$\{2\}$	%eax	0b 0000 0000 0100 0000 0001 0000 0100 0011
			0b 0000 0000 0100 0000 0001 0000 0100 0100
		%ebx	0b 0000 0000 0100 0000 0001 0000 000* *000

Table 1. VDs during analysis



```
start:
0x00:     jmp   0x18
0x02:     inc   %edi
0x03:     mov   $0x1, %edi
0x08:     cmpb  $0x2, %al
0x0a:     jle   0x10
0x0c:     nop
0x0d:     nop
0x0e:     nop
0x0f:     nop

l1:
0x10:     cmpb  $0x2, %al
0x12:     jnl   0x18
0x14:     nop
0x15:     nop
0x16:     nop
0x17:     nop

l2:
0x18:     shrl  $0x1e, %ebx
0x1b:     shll  $0x3, %ebx
0x1e:     addl  $0x00, %ebx
0x24:     incb  $0x00403000
0x2a:     cmpb  $0x2, 0x00403000
0x31:     je    0x35
0x33:     jmp   %ebx

l3:
0x35:     shrl  $0x1f, %eax
0x38:     addl  $0x43, %eax
0x3d:     cmpl  %eax, %ebx
0x3f:     je    $0x4a
0x41:     jmp   %eax
0x43:     nop
0x44:     addl  %ebx, %eax
0x46:     subl  %eax, %ebx
0x48:     jmp   $0x4e

l4:
0x4a:     addl  %eax, %ebx
0x4c:     subl  %ebx, %eax

l5:
0x4e:     xorl  %eax, %eax
0x50:     pushl $0x0
0x52:     call  0x58
0x58:     cmp   ExitProcesskernel32.dll
```

Figure 1. `demo2.exe` from Kinder [5], with control flow edges originating from jumps

References

[1] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, Aug. 1986. .

[2] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the Sixth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282. ACM Press, 1979.

[3] A. Gurfinkel and S. Chaki. Boxes: A symbolic abstract domain of boxes. In *Static Analysis*, LNCS(6337), pages 287–303. Springer, Jan. 2010. ISBN 978-3-642-15768-4, 978-3-642-15769-1. URL `http://link.springer.com/chapter/10.1007/978-3-642-15769-1_18`.

[4] J. Kinder. *Static analysis of x86 executables*. PhD thesis, Technische Universität Darmstadt, 2010.

[5] J. Kinder and D. Kravchenko. Alternating control flow reconstruction. In *Verification, Model Checking, and Abstract Interpretation*, LNCS(7148), pages 267–282. Springer, Jan. 2012. ISBN 978-3-642-27939-3, 978-3-642-27940-9.

[6] J. Kinder and H. Veith. Jakstab: A static analysis platform for binaries. In *Computer Aided Verification*, LNCS(5123), pages 423–427. Springer, Jan. 2008. ISBN 978-3-540-70543-7, 978-3-540-70545-1.

[7] J.-C. Madre and J.-P. Billon. Proving circuit correctness using formal comparison between expected and extracted behaviour. In *Proceedings of the 25th ACM/IEEE Design Automation Conference*, DAC '88, page 205–210. IEEE Computer Society Press, 1988. ISBN 0-8186-8864-5.

[8] A. Sepp, B. Mihaila, and A. Simon. Precise static analysis of binaries by extracting relational information. In *Proceedings of the 18th Working Conference on Reverse Engineering*, WCRE '11, page 357–366. IEEE Computer Society, 2011. ISBN 978-0-7695-4582-0. .