



A New Interval Approximation Supporting Bit Operations and Byte Access

Werner Backes

Department of Computer Science
Stevens Institute of Technology
Hoboken, NJ, 07030, USA
`wbackes@cs.stevens.edu`

Department of Computer Science
Technical Report CS-2006-4
June 6, 2006

Department of Computer Science · Stevens Institute of Technology
Castle Point on Hudson · Hoboken, NJ 07030 · USA

A New Interval Approximation Supporting Bit Operations and Byte Access

Werner Backes*

Abstract

In this paper we present a new variant of the commonly used interval approximation based on the so-called *valid interval* approach. This new approach supports arithmetic and bit operations including shift and rotate functions. Furthermore, it allows read and write access to the byte representation of integer and floating point values. We present the necessary functions for integer intervals in detail, including examples.

1 Introduction

The problem of finding bugs in programs exists as long as the computer itself. Nowadays larger programming projects often encompass different modules ranging from graphical user interfaces to embedded systems applications. For each module, the appropriate programming language is used and each module has to be tested in a certain runtime environment. These test phases take a substantial amount of the total development time for a software project.

Static analysis tools are used to verify safety properties of various programming languages. Without runtime tests, we can verify the absence of bugs like buffer overflow, division by zero or the use of uninitialized variables. This kind of errors often lead to exploitable security holes. Static analysis tools would be the perfect choice to eliminate the costly and time consuming runtime tests in larger software projects. But most of the analysis tools have problems dealing with under- and overflow of values and low-level operations like bit operations or byte access. These problems usually occur using low-level or assembler languages for e.g. drivers in operating systems or embedded systems. Our work helps in addressing these problems.

It is in this context, that we present a new variant of the commonly used interval approximation. In this paper we will focus on describing the new approach and not talk about the analysis using the new interval approximation. We believe that a detailed description will help in analyzing real-world applications or inspire others to modify or extend their abstract

*This work was done while the author was at the MPI, Saarbrücken, Germany.

domains. The new approach uses lists of valid intervals for the abstraction of sets of integer or floating point values. We introduce the notion of *valid intervals* and show in detail how arithmetic and bit operations can be implemented. We provide examples to illustrate the possible cases. In order to allow read and write access to the byte representation, we furthermore have to restrict the definition of valid intervals. Due to page limit we will in following only discuss integer intervals.

Outline: In section 2 we motivate the need for an abstract domains that support bit operations and byte access. Section 3 provides definitions and background for valid intervals and in section 4 we detail the arithmetic and bit operations. We restrict the definition for valid intervals in section 5 to allow read and write byte access. We discuss related work in section 6.

2 Motivation

Low-level bit operations like *and*, *or*, *xor* or *shift* operations are commonly used within operating systems, e.g. for device drivers, file systems, or network code. The following code fragment of function `piix_set_piomode` demonstrates a typical use of bit operations. The code was taken from the Intel PATA/SATA driver in Linux 2.6.15.1 (`drivers/scsi/ata_piix.c`).

```
(1)  ...
(2)  pci_read_config_word(dev, master_port, &master_data);
(3)  if (is_slave) {
(4)      master_data |= 0x4000;
(5)      /* enable PPE, IE and TIME */
(6)      master_data |= 0x0070;
(7)      pci_read_config_byte(dev, slave_port, &slave_data);
(8)      slave_data &= (ap->hard_port_no ? 0x0f : 0xf0);
(9)      slave_data |= (timings[pio][0] << 2) |
(10)                   (timings[pio][1] << (ap->hard_port_no ? 4 : 0));
(11) } else {
(12)     master_data &= 0xccf8; /* enable PPE, IE and TIME */
(13)     master_data |= 0x0007;
(14)     master_data |= (timings[pio][0] << 12) | (timings[pio][1] << 8);
(15) }
(16) pci_write_config_word(dev, master_port, master_data);
(17) ...
```

Multimedia applications that encode, decode or modify audio or video data usually need access to single or sequences of bits and therefore depend on bit operations. The following function `getbits_fast` is part of `mpg123` (`getbits.c`), a program that can be used to play different mpeg encoded audio files. The function extracts a number of bits from a given data stream.

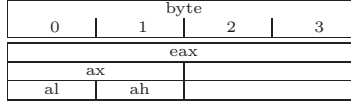
```
(1) unsigned int getbits_fast(struct bitstream_info *bitbuf,int number_of_bits)
(2) {
(3)     unsigned int rval;
(4)     rval = (unsigned char)(bitbuf->wordpointer[0] << bitbuf->bitindex);
(5)     rval |= ((unsigned int) bitbuf->wordpointer[1] << bitbuf->bitindex) >> 8;
(6)     rval <<= number_of_bits;
(7)     rval >>= 8;
(8)     bitbuf->bitindex += number_of_bits;
(9)     bitbuf->wordpointer += (bitbuf->bitindex >> 3);
```

```

(10) bitbuf->bitindex += 7;
(11) return rval;
(12) }

```

Read and write byte access is necessary to analyze assembler code. For examples, the x86 architecture uses subregisters to access a certain part of a register. The following picture shows the 4 byte register *eax* and its subregisters *ax*, *al*, *ah*.



The presented valid interval approach supports all operations needed for the analysis of low-level code. We now detail this approach.

3 Intervals

Definition 1 We call $[a, b]$ with $a, b \in R$ and $a \leq b$ an interval. A value $x \in R$ is member of the interval $x \in [a, b]$ iff $a \leq x \leq b$.

We, like most assembler languages, do not distinguish between signed and unsigned integers. The signed or unsigned use of a value is dependent on the operation we are performing. The following table shows bit strings and their signed and unsigned interpretation for integers of byte size n .

$b_0 \dots b_{8n-1}$	unsigned		signed	
00 ... 00	0	0	0	0
10 ... 00	1	1	1	1
...
11 ... 10	$2^{8n-1} - 1$	$\frac{m}{2} - 1$	$2^{8n-1} - 1$	$\frac{m}{2} - 1$
00 ... 01	2^{8n-1}	$\frac{m}{2}$	-2^{8n-1}	$-\frac{m}{2}$
10 ... 01	$2^{8n-1} + 1$	$\frac{m}{2} + 1$	$-2^{8n-1} + 1$	$-\frac{m}{2} + 1$
...
01 ... 11	$2^{8n} - 2$	$m - 2$	-2	-2
11 ... 11	$2^{8n} - 1$	$m - 1$	-1	-1

The comparison of two bit strings is dependent on the signed or unsigned interpretation of bit strings. Therefore, Definition 1 does not work for bit strings.

Example 2 For the bit strings $11 \dots 10$ and $00 \dots 10$ the following holds for unsigned interpretation $11 \dots 10 < 00 \dots 10$. Using the signed interpretation of bit strings $11 \dots 10 > 00 \dots 10$ holds.

Looking at the two forms of interpretations, we notice that bit strings behave like \mathbb{Z}_m , an euclidean ring with 1. The interpretations can be compared to the problem of choosing a member of a residue class $\bar{a} \in \mathbb{Z}_m$.

Definition 3 For a residue class $\bar{a} \in \mathbb{Z}_m$ we call $a^+ \in \mathbb{Z}$ the smallest non-negative member and $a^- \in \mathbb{Z}$ the member of the residue class with the smallest absolute value. For $a', a'' \in \bar{a}$ with $|a'| = |a''|$ and $a' < a''$ we define $a^- \stackrel{\text{def}}{=} a'$.

Example 4 For \mathbb{Z}_8 the set of smallest non-negative members of the residue classes is $\{0, 1, \dots, 7\}$. The set of the members with minimal absolute value is $\{-4, \dots, -1, 0, \dots, 3\}$. We take -4 instead of 4 due to Definition 3 as member with minimal absolute value. \mathbb{Z}_8 can be written as $\{\bar{0}, \bar{1}, \dots, \bar{7}\}$ or $\{-\bar{4}, \dots, -\bar{1}, \bar{0}, \dots, \bar{3}\}$.

We now modify definition 1 for $R = \mathbb{Z}_m$.

Definition 5 The interval $[\bar{a}, \bar{b}]$ with $\bar{a}, \bar{b} \in \mathbb{Z}_m$ and $a^+ \leq b^+$ denotes the set of all $\bar{x} \in \mathbb{Z}_m$ with $a^+ \leq x^+$ and $x^+ \leq b^+$. We call $I(\mathbb{Z}_m)$ the set of all intervals over \mathbb{Z}_m with $I(\mathbb{Z}_m) \stackrel{\text{def}}{=} \{\perp\} \cup \{[\bar{a}, \bar{b}] \mid [\bar{a}, \bar{b}] \text{ is interval and } \bar{a}, \bar{b} \in \mathbb{Z}_m\}$, where \perp denotes the empty and \top the interval containing all elements of \mathbb{Z}_m .

The operations addition, subtractions and multiplication are independent of the choice of the member of the residue class because \mathbb{Z}_m is an euclidean ring with 1. We provide two pseudo-divisions for signed and unsigned interpretations of bit strings. We have to modify Definition 1 because intervals $[\bar{a}, \bar{b}]$ exist with $a^+ < b^+$ but $a^- > b^-$ which may cause problems for the division.

Example 6 Given an interval $[\bar{a}, \bar{b}]$, $\bar{a}, \bar{b} \in \mathbb{Z}_8$ with $\bar{a} = \bar{2}$ and $\bar{b} = \bar{6}$. For this interval $a^+ < b^+ \Leftrightarrow 2 < 6$ holds, but $a^- > b^- \Leftrightarrow 2 > -2$ as well.

Definition 7 We call $[\bar{a}, \bar{b}]$ with $\bar{a}, \bar{b} \in \mathbb{Z}_m$ and $a^+ \leq b^+$ a valid interval, iff $a^+ \geq \frac{m}{2}$ or $b^+ < \frac{m}{2}$ holds. We call $I^*(\mathbb{Z}_m)$ the set of all valid intervals over \mathbb{Z}_m with $I^*(\mathbb{Z}_m) \stackrel{\text{def}}{=} \{\perp\} \cup \{[\bar{a}, \bar{b}] \mid [\bar{a}, \bar{b}] \text{ is valid interval}\}$

Remark 8 The interval $[\bar{a}, \bar{b}]$ is valid, iff $a^+ \leq b^+$ and $a^- \leq b^-$ holds.

The additional constraints for valid intervals simplify the basic operations but also lead to the introduction of lists of valid intervals, since no valid interval exists that covers \mathbb{Z}_m . The following lemma describes the conversion from intervals to valid intervals.

Lemma 9 An interval $[\bar{a}, \bar{b}]$ is valid or can be split into two valid intervals that contain every element of $[\bar{a}, \bar{b}]$.

Proof: Construction: $[\bar{a}, \bar{b}]$ not valid.

$$[\bar{a}, \bar{b}] \Rightarrow \begin{cases} [\bar{a}_1, \bar{b}_1] \stackrel{\text{def}}{=} [\bar{a}, \frac{m}{2} - \bar{1}] \\ [\bar{a}_2, \bar{b}_2] \stackrel{\text{def}}{=} [\frac{m}{2}, \bar{b}] \end{cases}$$

The intervals $[\bar{a}_1, \bar{b}_1]$ and $[\bar{a}_2, \bar{b}_2]$ are valid and contain every element of $[\bar{a}, \bar{b}]$. ■

Lists of valid intervals can be used for the approximation of sets of possible values. The list length is bounded by a parameter, which allows us to increase the precision of the approximation and analysis.

Definition 10 We call $([\bar{a}_i, \bar{b}_i])_{i=1}^n$ a list of valid intervals $[\bar{a}_i, \bar{b}_i]$ with n the maximal length. $\bar{x} \in \mathbb{Z}_m$ is member of the list of intervals $\bar{x} \in ([\bar{a}_i, \bar{b}_i])_{i=1}^n$, iff $\exists i$ with $\bar{x} \in [\bar{a}_i, \bar{b}_i]$. With $I_n^*(\mathbb{Z}_m)$ we denote the set of all lists of valid intervals over \mathbb{Z}_m with maximal length n .

4 Operations

For the implementation of arithmetic operations on lists of valid intervals we are using operations defined for single valid intervals. The algorithm `merge` reduces the number of intervals and converts intervals to valid intervals.

For $op \in \{add, sub, mul, div, udiv, mod, umod\}$:

$$op : I_n^*(\mathbb{Z}_m) \times I_n^*(\mathbb{Z}_m) \longrightarrow I_n^*(\mathbb{Z}_m)$$

$$op \left(([\bar{a}_i, \bar{b}_i])_{i=1}^n, ([\bar{c}_j, \bar{d}_j])_{j=1}^n \right) \stackrel{\text{def}}{=} \text{merge} \left((op([\bar{a}_i, \bar{b}_i], [\bar{c}_j, \bar{d}_j]))_{i,j=1}^n \right)$$

For the algorithm `merge` we have to define the distance of two intervals.

Definition 11 For two intervals $[\bar{a}, \bar{b}]$ and $[\bar{c}, \bar{d}]$ with $\bar{a}, \bar{b}, \bar{c}, \bar{d} \in \mathbb{Z}_m$ and $a^+ \leq c^+$ we define the distance of $[\bar{a}, \bar{b}]$ and $[\bar{c}, \bar{d}]$ as follows:

$$\text{distance} : I(\mathbb{Z}_m) \times I(\mathbb{Z}_m) \longrightarrow \mathbb{Z}$$

$$\text{distance}([\bar{a}, \bar{b}], [\bar{c}, \bar{d}]) \stackrel{\text{def}}{=} \begin{cases} 0, & \text{if } c^+ \leq b^+ \\ c^+ - b^+ & \text{otherwise} \end{cases}$$

The algorithm `merge` works as follows:

INPUT: L : list of intervals

OUTPUT: L : list of valid intervals with maximal length n

- (1) sort L ascending according to the lower bounds
- (2) **foreach** ($k \in L$) **do**
- (3) **foreach** ($l \in L$) **do**
- (4) **if** ($k \cap l \neq \emptyset$) **then**
- (5) $j := k \cup l$; $L := L \setminus \{k, l\}$; $L := L \cup \{j\}$
- (6) **while** ($|L| > \frac{n}{2}$) **do**
- (7) find i with $\text{distance}([\bar{a}_i, \bar{b}_i], [\bar{a}_{i+1}, \bar{b}_{i+1}])$ minimal
- (8) $L := L \setminus \{[\bar{a}_i, \bar{b}_i], [\bar{a}_{i+1}, \bar{b}_{i+1}]\}$
- (9) $L := L \cup \{[\bar{a}_i, \bar{b}_{i+1}]\}$

- (10) **foreach** ($l \in L$) **do**
(11) replace l with valid intervals

The algorithm **merge** constructs a list of valid intervals with maximal length n out of a list of intervals. We loose precision when two intervals with minimal distance are replaced by a single interval.

4.1 Arithmetic Operations

We now define arithmetic operations on valid intervals. The resulting intervals are not necessarily valid. The operations addition and subtractions are defined as follows:

$$\{add, sub\} : I^*(\mathbb{Z}_m) \times I^*(\mathbb{Z}_m) \longrightarrow I(\mathbb{Z}_m) \times I(\mathbb{Z}_m)$$

$$add([\bar{a}, \bar{b}], [\bar{c}, \bar{d}]) \stackrel{\text{def}}{=} \begin{cases} ([\overline{a^+ + c^+}, \overline{b^+ + d^+}], \perp) \\ \text{if } (a^+ + c^+ \bmod m) \leq (b^+ + d^+ \bmod m) \\ ([\overline{a^+ + c^+}, \overline{m-1}], [\bar{0}, \overline{b^+ + d^+}]) \text{ otherwise} \end{cases}$$

$$sub([\bar{a}, \bar{b}], [\bar{c}, \bar{d}]) \stackrel{\text{def}}{=} \begin{cases} ([\overline{a^+ - d^+}, \overline{b^+ - c^+}], \perp) \\ \text{if } (a^+ - d^+ \bmod m) \leq (b^+ - c^+ \bmod m) \\ ([\overline{a^+ - d^+}, \overline{m-1}], [\bar{0}, \overline{b^+ - c^+}]) \text{ otherwise} \end{cases}$$

We have to deal with the overflow and underflow of values. For a double overflow of the addition, $a^+ + c^+ \geq m$ and a double underflow of the subtractions, $b^+ - c^+ < 0$ we get only one resulting interval.

Example 12 The following examples demonstrate the possible cases for addition and subtraction for valid intervals over \mathbb{Z}_{16} .

$$\begin{aligned} add([\bar{0}, \bar{2}], [\bar{1}, \bar{3}]) &\stackrel{\text{def}}{=} ([\bar{1}, \bar{5}], \perp) && (0 + 1 \bmod 16) \leq (2 + 3 \bmod 16) \\ add([\bar{9}, \bar{12}], [\bar{2}, \bar{6}]) &\stackrel{\text{def}}{=} ([\bar{11}, \bar{15}], [\bar{0}, \bar{2}]) && (9 + 2 \bmod 16) > (12 + 6 \bmod 16) \\ sub([\bar{13}, \bar{15}], [\bar{9}, \bar{11}]) &\stackrel{\text{def}}{=} ([\bar{2}, \bar{6}], \perp) && (13 - 11 \bmod 16) \leq (15 - 9 \bmod 16) \\ sub([\bar{9}, \bar{12}], [\bar{10}, \bar{13}]) &\stackrel{\text{def}}{=} ([\bar{12}, \bar{15}], [\bar{0}, \bar{4}]) && (9 - 13 \bmod 16) > (12 - 10 \bmod 16) \end{aligned}$$

The following two examples demonstrate the double over- and underflow:

$$\begin{aligned} add([\bar{11}, \bar{14}], [\bar{6}, \bar{7}]) &\stackrel{\text{def}}{=} ([\bar{1}, \bar{5}], \perp) && (11 + 6 \bmod 16) \leq (14 + 7 \bmod 16) \\ sub([\bar{1}, \bar{2}], [\bar{9}, \bar{11}]) &\stackrel{\text{def}}{=} ([\bar{6}, \bar{9}], \perp) && (1 - 11 \bmod 16) \leq (2 - 9 \bmod 16) \end{aligned}$$

Note that the resulting interval $[\bar{6}, \bar{9}]$ is not valid.

For the multiplication of valid intervals we have to take a closer look at the degree of overflow. The multiplication is defined as follows:

$$mul : I^*(\mathbb{Z}_m) \times I^*(\mathbb{Z}_m) \longrightarrow I(\mathbb{Z}_m) \times I(\mathbb{Z}_m)$$

$$mul \left([\bar{a}, \bar{b}], [\bar{c}, \bar{d}] \right) \stackrel{\text{def}}{=} \begin{cases} ([\bar{0}, \overline{m-1}], \perp) & \text{if } degree > 1 \\ ([\bar{0}, \overline{b^+ \cdot d^+}], [\overline{a^+ \cdot c^+}, \overline{m-1}]) & \text{if } degree = 1 \\ ([\overline{a^+ \cdot c^+}, \overline{b^+ \cdot d^+}], \perp) & \text{otherwise} \end{cases}$$

with $degree \stackrel{\text{def}}{=} \lfloor \frac{b^+ \cdot d^+}{m} \rfloor - \lfloor \frac{a^+ \cdot c^+}{m} \rfloor$ the degree of overflow of the multiplication. For a degree of overflow of 1 we get two resulting intervals like for addition and subtractions.

Example 13 These examples show the possible cases of overflow for the multiplication of valid intervals over \mathbb{Z}_{16} .

$$mul \left([\bar{0}, \bar{4}], [\bar{10}, \bar{14}] \right) \stackrel{\text{def}}{=} ([\bar{0}, \bar{15}], \perp) \quad degree \stackrel{\text{def}}{=} \left\lfloor \frac{4 \cdot 14}{16} \right\rfloor - \left\lfloor \frac{0 \cdot 10}{16} \right\rfloor = 3$$

$$mul \left([\bar{2}, \bar{5}], [\bar{3}, \bar{4}] \right) \stackrel{\text{def}}{=} ([\bar{0}, \bar{4}], [\bar{6}, \bar{15}]) \quad degree \stackrel{\text{def}}{=} \left\lfloor \frac{5 \cdot 4}{16} \right\rfloor - \left\lfloor \frac{2 \cdot 3}{16} \right\rfloor = 1$$

$$mul \left([\bar{1}, \bar{3}], [\bar{2}, \bar{4}] \right) \stackrel{\text{def}}{=} ([\bar{2}, \bar{12}], \perp) \quad degree \stackrel{\text{def}}{=} \left\lfloor \frac{3 \cdot 4}{16} \right\rfloor - \left\lfloor \frac{1 \cdot 2}{16} \right\rfloor = 0$$

We define two types of pseudo divisions the unsigned version *udiv* and the signed version *div*. They are defined as follows:

$$\{udiv, div\} : I^*(\mathbb{Z}_m) \times I^*(\mathbb{Z}_m) \longrightarrow I(\mathbb{Z}_m)$$

$$udiv \left([\bar{a}, \bar{b}], [\bar{c}, \bar{d}] \right) \stackrel{\text{def}}{=} \begin{cases} \text{undefined}, & \text{if } c^+ = 0 \\ [\lfloor \frac{a^+}{d^+} \rfloor, \lfloor \frac{b^+}{c^+} \rfloor] & \text{otherwise} \end{cases}$$

$$div \left([\bar{a}, \bar{b}], [\bar{c}, \bar{d}] \right) \stackrel{\text{def}}{=} \begin{cases} \text{undefined}, & \text{if } c^- \leq 0 \leq d^- \\ [\lfloor \frac{a^-}{d^-} \rfloor, \lfloor \frac{b^-}{c^-} \rfloor] & \text{if } (a^- \geq 0) \wedge (c^- \geq 0) \\ [\lfloor \frac{b^-}{c^-} \rfloor, \lfloor \frac{a^-}{d^-} \rfloor] & \text{if } (a^- < 0) \wedge (c^- < 0) \\ [\lfloor \frac{b^-}{d^-} \rfloor, \lfloor \frac{a^-}{c^-} \rfloor] & \text{if } (a^- \geq 0) \wedge (c^- < 0) \\ [\lfloor \frac{a^-}{c^-} \rfloor, \lfloor \frac{b^-}{d^-} \rfloor] & \text{if } (a^- < 0) \wedge (c^- \geq 0) \end{cases}$$

For *div* we only have to compare a^- and c^- , since $[\bar{a}, \bar{b}]$ and $[\bar{c}, \bar{d}]$ are valid intervals. From the definition of valid intervals ($b^- \geq 0$) follows from ($a^- \geq 0$) and ($b^- < 0$) from ($a^- < 0$).

Example 14 The following examples demonstrate all possible cases for the signed and unsigned pseudo-division *div* and *udiv* for valid intervals over \mathbb{Z}_{16} .

$$udiv \left([\bar{8}, \bar{13}], [\bar{2}, \bar{5}] \right) \stackrel{\text{def}}{=} \left[\left\lfloor \frac{8}{5} \right\rfloor, \left\lfloor \frac{13}{2} \right\rfloor \right] = [\bar{1}, \bar{6}]$$

$$\begin{aligned}
\text{div}([\bar{2}, \bar{5}], [\bar{3}, \bar{4}]) &\stackrel{\text{def}}{=} \left[\left[\frac{2}{4}, \frac{5}{3} \right] \right] = [\bar{0}, \bar{1}] \\
\text{div}([\bar{-5}, \bar{-3}], [\bar{-2}, \bar{-1}]) &\stackrel{\text{def}}{=} \left[\left[\frac{-3}{-2}, \frac{-5}{-1} \right] \right] = [\bar{1}, \bar{5}] \\
\text{div}([\bar{-7}, \bar{-4}], [\bar{3}, \bar{4}]) &\stackrel{\text{def}}{=} \left[\left[\frac{-7}{3}, \frac{-4}{4} \right] \right] = [\bar{-3}, \bar{-1}] \\
\text{div}([\bar{4}, \bar{6}], [\bar{-7}, \bar{-3}]) &\stackrel{\text{def}}{=} \left[\left[\frac{6}{-3}, \frac{4}{-7} \right] \right] = [\bar{-2}, \bar{-1}]
\end{aligned}$$

We also support the corresponding modulo operations mod and $umod$. For the modulo operation mod the result of $x^- \bmod y^-$ is negative if $x^- < 0$ and y^- not a divisor of x^-

$$\begin{aligned}
\{mod, umod\} : I^*(\mathbb{Z}_m) \times I^*(\mathbb{Z}_m) &\longrightarrow I(\mathbb{Z}_m) \\
mod([\bar{a}, \bar{b}], [\bar{c}, \bar{d}]) &\stackrel{\text{def}}{=} \begin{cases} \text{undefined if } c^- \leq 0 \leq d^- \\ [\bar{a}, \bar{b}] \text{ if } (b^- < c^-) \wedge (a^- \geq 0) \wedge (c^- \geq 0) \\ \quad \text{or } (a^- < d^-) \wedge (a^- < 0) \wedge (c^- < 0) \\ [\bar{0}, \overline{d^- - 1}] \text{ if } (a^- \geq 0) \wedge (c^- \geq 0) \\ [\bar{0}, \overline{|c^-| - 1}] \text{ if } (a^- \geq 0) \wedge (c^- < 0) \\ [\overline{-(d^-) + 1}, \bar{0}] \text{ if } (a^- < 0) \wedge (c^- \geq 0) \\ [\overline{c^- + 1}, \bar{0}] \text{ if } (a^- < 0) \wedge (c^- < 0) \end{cases} \\
umod([\bar{a}, \bar{b}], [\bar{c}, \bar{d}]) &\stackrel{\text{def}}{=} \begin{cases} \text{undefined if } c^+ = 0 \\ [\bar{a}, \bar{b}] \text{ if } b^+ < c^+ \\ [\bar{0}, \bar{b}^+] \text{ if } (b^+ \geq c^+) \wedge (b^+ < d^+) \\ [\bar{0}, \overline{d^+ - 1}] \text{ if } (b^+ \geq c^+) \wedge (b^+ \geq d^+) \end{cases}
\end{aligned}$$

Example 15 For the operations $umod$ and mod on valid intervals over \mathbb{Z}_{16} the following examples show all possible cases.

$$\begin{aligned}
mod([\bar{-3}, \bar{-2}], [\bar{-6}, \bar{-4}]) &\stackrel{\text{def}}{=} [\bar{-3}, \bar{-2}] & umod([\bar{1}, \bar{7}], [\bar{8}, \bar{9}]) &\stackrel{\text{def}}{=} [\bar{1}, \bar{7}] \\
mod([\bar{1}, \bar{4}], [\bar{5}, \bar{7}]) &\stackrel{\text{def}}{=} [\bar{1}, \bar{4}] & umod([\bar{0}, \bar{7}], [\bar{3}, \bar{5}]) &\stackrel{\text{def}}{=} [\bar{0}, \bar{4}] \\
mod([\bar{4}, \bar{7}], [\bar{3}, \bar{5}]) &\stackrel{\text{def}}{=} [\bar{0}, \bar{4}] & umod([\bar{1}, \bar{5}], [\bar{2}, \bar{6}]) &\stackrel{\text{def}}{=} [\bar{0}, \bar{5}] \\
\\
mod([\bar{1}, \bar{5}], [\bar{-4}, \bar{-2}]) &\stackrel{\text{def}}{=} [\bar{0}, \bar{3}] \\
mod([\bar{-6}, \bar{-3}], [\bar{2}, \bar{4}]) &\stackrel{\text{def}}{=} [\bar{-3}, \bar{0}] \\
mod([\bar{-5}, \bar{-2}], [\bar{-3}, \bar{-1}]) &\stackrel{\text{def}}{=} [\bar{-2}, \bar{0}]
\end{aligned}$$

4.2 Bit Operations

The bit operations *shift_left* and *shift_right* behave like a multiplication and a unsigned division without overflow checking. They are defined the following way:

$$\begin{aligned}
 \text{shift_right} & : I^*(\mathbb{Z}_m) \times I^*(\mathbb{Z}_m) \longrightarrow I(\mathbb{Z}_m) \\
 \text{shift_left} & : I^*(\mathbb{Z}_m) \times I^*(\mathbb{Z}_m) \longrightarrow I(\mathbb{Z}_m) \times I(\mathbb{Z}_m) \\
 \text{shift_right}([\bar{a}, \bar{b}], [\bar{c}, \bar{d}]) & \stackrel{\text{def}}{=} \left[\left\lceil \frac{a^+}{2^{d^+}} \right\rceil, \left\lceil \frac{b^+}{2^{c^+}} \right\rceil \right] \\
 \text{shift_left}([\bar{a}, \bar{b}], [\bar{c}, \bar{d}]) & \stackrel{\text{def}}{=} \begin{cases} ([\bar{0}, \bar{0}], \perp) & \text{if } a^+ \cdot 2^{c^+} \geq m \\ ([\overline{a^+ \cdot 2^{c^+}}, \overline{m-1}], [\bar{0}, \bar{0}]) & \\ \quad \text{if } (a^+ \cdot 2^{c^+} < m) \wedge (b^+ \cdot 2^{d^+} \geq m) & \\ ([\overline{a^+ \cdot 2^{c^+}}, \overline{b^+ \cdot 2^{d^+}}], \perp) & \text{otherwise} \end{cases}
 \end{aligned}$$

Example 16 The following examples demonstrate the bit operations *shift_left* and *shift_right*. All possible cases for valid intervals over \mathbb{Z}_{16} are shown.

$$\begin{aligned}
 \text{shift_left}^\#([\bar{5}, \bar{7}], [\bar{3}, \bar{4}]) & \stackrel{\text{def}}{=} ([\bar{0}, \bar{0}], \perp) \\
 \text{shift_left}^\#([\bar{1}, \bar{3}], [\bar{2}, \bar{5}]) & \stackrel{\text{def}}{=} ([\bar{4}, \bar{15}], [\bar{0}, \bar{0}]) \\
 \text{shift_left}^\#([\bar{2}, \bar{3}], [\bar{1}, \bar{2}]) & \stackrel{\text{def}}{=} ([\bar{4}, \bar{12}], \perp) \\
 \text{shift_right}^\#([\bar{2}, \bar{3}], [\bar{3}, \bar{5}]) & \stackrel{\text{def}}{=} [\bar{0}, \bar{0}] \\
 \text{shift_right}^\#([\bar{1}, \bar{7}], [\bar{1}, \bar{2}]) & \stackrel{\text{def}}{=} [\bar{0}, \bar{3}] \\
 \text{shift_right}^\#([\bar{8}, \bar{11}], [\bar{0}, \bar{2}]) & \stackrel{\text{def}}{=} [\bar{2}, \bar{11}]
 \end{aligned}$$

For the implementation of the other bit operations we need a second approximation step. We have to convert valid intervals into a vector of possible bits.

$$\begin{aligned}
 \text{bitvector} & : I^*(\mathbb{Z}_m) \longrightarrow P(\{0, 1\}^n) \\
 \text{bitvector}([\bar{a}, \bar{b}]) & \stackrel{\text{def}}{=} \begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix}
 \end{aligned}$$

with a_1, \dots, a_n and b_1, \dots, b_n the bit representation of a and b and

$$c_i \stackrel{\text{def}}{=} \begin{cases} \{0, 1\}, & \text{if } (b^+ - a^+) \geq (2^i - 2^{i-1}) \\ a_i \cup b_i & \text{otherwise.} \end{cases}$$

The bit operations *and*, *or*, *xor* and *not* are defined as follows:

$$\begin{aligned}
 \{\text{and}, \text{or}, \text{xor}\} & : P(\{0, 1\}^n) \times P(\{0, 1\}^n) \longrightarrow P(\{0, 1\}^n) \\
 \text{not} & : P(\{0, 1\}^n) \longrightarrow P(\{0, 1\}^n)
 \end{aligned}$$

$$\begin{aligned}
and \left(\begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix}, \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix} \right) &\stackrel{\text{def}}{=} \begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix} \text{ with } c_i \stackrel{\text{def}}{=} \begin{cases} \{0\}, & \text{if } a_i = \{0\} \text{ or } b_i = \{0\} \\ \{1\}, & \text{if } a_i = \{1\} \text{ and } b_i = \{1\} \\ \{0, 1\}, & \text{otherwise} \end{cases} \\
or \left(\begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix}, \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix} \right) &\stackrel{\text{def}}{=} \begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix} \text{ with } c_i \stackrel{\text{def}}{=} \begin{cases} \{0\}, & \text{if } a_i = \{0\} \text{ and } b_i = \{0\} \\ \{1\}, & \text{if } a_i = \{1\} \text{ or } b_i = \{1\} \\ \{0, 1\}, & \text{otherwise} \end{cases} \\
xor \left(\begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix}, \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix} \right) &\stackrel{\text{def}}{=} \begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix} \text{ with } c_i \stackrel{\text{def}}{=} \begin{cases} \{0\} & \text{if } a_i = \{0\} \text{ and } b_i = \{0\} \\ & \text{or } a_i = \{1\} \text{ and } b_i = \{1\} \\ \{1\} & \text{if } a_i = \{0\} \text{ and } b_i = \{1\} \\ & \text{or } a_i = \{1\} \text{ and } b_i = \{0\} \\ \{0, 1\} & \text{otherwise} \end{cases} \\
not \left(\begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix} \right) &\stackrel{\text{def}}{=} \begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix} \text{ with } c_i \stackrel{\text{def}}{=} \begin{cases} \{0\}, & \text{if } a_i = \{1\} \\ \{1\}, & \text{if } a_i = \{0\} \\ \{0, 1\}, & \text{otherwise} \end{cases}
\end{aligned}$$

The functions *rotate_left* and *rotate_right* are only implemented for fixed values. For other values we approximate the result with \top .

$$\{rotate_left, rotate_right\} : P(\{0, 1\}^n) \times I^*(\mathbb{Z}_m) \longrightarrow P(\{0, 1\}^n)$$

$$rotate_left \left(\begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix}, [\bar{b}, \bar{b}] \right) \stackrel{\text{def}}{=} \begin{pmatrix} d_1 \\ \vdots \\ d_n \end{pmatrix}$$

$$rotate_right \left(\begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix}, [\bar{b}, \bar{b}] \right) \stackrel{\text{def}}{=} \begin{pmatrix} e_1 \\ \vdots \\ e_n \end{pmatrix}$$

with $d_i = a_{((i+b+1) \bmod n)+1}$ and $e_i = a_{((i-b+1) \bmod n)+1}$ and $a_i \neq \{0, 1\}$ for $1 \leq i \leq n$.

5 Byte Access

Another difficulty is allowing read and write access to the byte representation of integer values. For an integer value $x \in \mathbb{Z}_m$ with bit string x_0, \dots, x_{8n-1} , the function rep_n can easily compute the byte representation of x :

$$rep_n : \mathbb{Z}_m \longrightarrow (\mathbb{Z}_{256})^n$$

$$rep_n(\bar{x}) \stackrel{\text{def}}{=} \bar{b}_0, \dots, \bar{b}_{n-1} \quad \text{with } b_i = \sum_{j=8i}^{8i+7} x_j 2^{j-8i}$$

We now extend function rep_n to valid intervals using the byte representation of the lower and upper bound of the valid interval. It computes a vector of byte intervals of dimension n for a valid interval:

$$rep_n : I^*(\mathbb{Z}_m) \longrightarrow (\mathbb{Z}_{256})^n$$

$$rep_n([\bar{a}, \bar{b}]) \stackrel{\text{def}}{=} [\bar{c}_0, \bar{d}_0], \dots, [\bar{c}_{n-1}, \bar{d}_{n-1}]$$

$$c_i \stackrel{\text{def}}{=} \begin{cases} \bar{0}, & \text{if } b^+ - a^+ \geq 2^{8(i+1)} - 2^{8i} \\ \bar{a}_i, & \text{otherwise} \end{cases} \quad d_i \stackrel{\text{def}}{=} \begin{cases} \bar{255}, & \text{if } b^+ - a^+ \geq 2^{8(i+1)} - 2^{8i} \\ \bar{b}_i, & \text{otherwise} \end{cases}$$

To allow an easier reconstruction of the interval using a vector of byte intervals, we require that for every byte of the vector it holds that $c_i^+ \leq d_i^+$. Obviously, this is not true for all valid intervals in Definition 7.

Example 17 $[10, 256] \in I^*(\mathbb{Z}_{256})$ has representation $[10, 0][0, 1]$ with $c_0^+ > d_0^+$.

We now have to modify the definition of valid intervals to allow access to the byte representation and simplify the implementation.

Definition 18 We call an interval $[a, b]$ with $\bar{a}, \bar{b} \in \mathbb{Z}_m$ with $rep_n([\bar{a}, \bar{b}]) \stackrel{\text{def}}{=} [c_0, d_0], \dots, [c_{n-1}, d_{n-1}]$ valid, if it is valid according to Definition 7 and for every byte $c_i^+ \leq d_i^+$ with $0 \leq i < n$ holds.

We are able to split intervals into valid intervals according to Definition 18.

Lemma 19 An interval $[\bar{a}, \bar{b}]$ with $\bar{a}, \bar{b} \in \mathbb{Z}_m$ is valid or can be split into four valid intervals that contain all elements of $[\bar{a}, \bar{b}]$.

Proof: Construction: $[\bar{a}, \bar{b}]$ with $rep_n([\bar{a}, \bar{b}]) \stackrel{\text{def}}{=} [c_0, d_0], \dots, [c_{n-1}, d_{n-1}]$ valid according to Definition 7 but not to Definition 18.

$$p \stackrel{\text{def}}{=} \max\{i \mid c_i > d_i\}$$

$$\bar{a}' \stackrel{\text{def}}{=} \sum_{j=0}^i 255 \cdot 2^{8j} + \sum_{j=i+1}^{m-1} c_j \cdot 2^{8j}$$

$$\bar{b}' \stackrel{\text{def}}{=} \sum_{j=0}^i 0 \cdot 2^{8j} + (c_{i+1} + 1) \cdot 2^{(i+1)8} + \sum_{j=i+2}^{m-1} c_j \cdot 2^{8j}$$

$[\bar{a}, \bar{a}']$ and $[\bar{b}', \bar{b}]$ are valid according Definition 18 due to the construction of \bar{a}' and \bar{b}' . For every byte $a_j^+ \leq a_j'^+$ and $b_j'^+ \leq b_j^+$ holds due to the byte representation of a, a', b', b . $[\bar{a}, \bar{a}']$ and $[\bar{b}', \bar{b}]$ contains all elements of $[\bar{a}, \bar{b}]$ because $b'^+ = a'^+ + 1$. ■

We now can implement rep_n^{-1} for intervals:

$$\begin{aligned}
 rep_n^{-1} &: (\mathbb{Z}_{256})^n \longrightarrow \mathbb{Z}_m \\
 rep_n^{-1} &([\bar{a}_0, \bar{b}_0], \dots, [\bar{a}_{m-1}, \bar{b}_{m-1}]) \stackrel{\text{def}}{=} [\bar{a}, \bar{b}] \\
 a^+ &\stackrel{\text{def}}{=} \sum_{i=1}^{n-1} a_i^+ \cdot 2^{8i} \quad \text{and} \quad b^+ \stackrel{\text{def}}{=} \sum_{i=1}^{n-1} b_i^+ \cdot 2^{8i}
 \end{aligned}$$

If $a_i^+ \leq b_i^+$ for $0 \leq i < m$ holds then $a^+ \leq b^+$ for the resulting values $\bar{a}, \bar{b} \in \mathbb{Z}_m$. $[\bar{a}, \bar{b}]$ is an interval. We can extend the functions rep_n and rep_n^{-1} to lists of valid intervals using the algorithm `merge`.

6 Related Work

In this section we present only a small selection of analysis tools or approaches and how valid intervals can be used to increase the quality of the analysis.

Model Checking: Model checking is a technique for verifying finite state concurrent systems. Counterexamples are produced to demonstrate undesirable behavior. The problem of model checking is to deal with the state space explosion problem. Analysis tools like *SLAM* [4, 5, 24] and *BLAST* [14, 21] use a model checking approach with predicate abstraction for analyzing C programs. *SLAM* produces an abstract boolean program out of a C program with a fixed number of predicates, using boolean variables for each predicate. *BLAST* is using counterexamples for the automatic abstraction refinement to construct an abstract model. Lazy abstraction is used to avoid the state space explosion. The abstract model is constructed and refined if necessary. But predicate abstraction is inappropriate for byte access or bit operations. We can use valid intervals for example to determine which of the predicates is affected by the modification of the byte representation of an integer or floating point value.

Intermediate Languages: Valid intervals can be used to analyze any intermediate or assembler language, like a simplified C programming language for *CCured/CIL* [16, 17, 23] or the x86 assembler language for *CodeSurfer/x86* [2, 3]. *CCured/CIL* adds runtime checks to the program according to the analysis. The program halts on errors but the 10% to 60% overhead for these runtime checks are not acceptable for some applications. Modifications to the source program are often necessary. The valid interval approach could be used to reduce the number of runtime checks which could reduce the resulting overhead to an acceptable rate. *CodeSurfer/x86* extracts the model from an x86 executable and uses reachability algorithms to verify properties for this the model. A static analysis algorithm called value-set analysis has been implemented. We developed this approach to analyze XRTL, a low-level intermediate language generated by a modified GCC [1].

7 Conclusion

In this paper we developed a variant of the commonly used interval approximation. Lists of valid intervals are used to approximate sets of possible values. We have detailed the valid interval approach by defining the notion of valid intervals for integers. Arithmetic operations were introduced that deal with the under- and overflow of integer values. We showed in detail how to implement bit operations including shift and rotate functions. After introducing a refined definition of valid intervals we were even able to handle read and write access to the byte representation of integers. We are planning to incorporate the strided interval approach [19] to increase the precision of the analysis. A future application would be the integration of the valid interval approach into other analysis tools.

References

- [1] W. Backes. *Programmanalyse des XRTL Zwischencodes*. (In German). PhD thesis, Universität des Saarlandes, 2005.
- [2] G. Balakrishnan and T. W. Reps. Analyzing Memory Accesses in x86 Executables. In *Proceedings of Conference on Compiler Construction*, volume 2985 of *LNCS*, pages 5–23. Springer, 2004.
- [3] G. Balakrishnan, R. Gruian, T. W. Reps, and T. Teitelbaum. Codesurfer/x86-A Platform for Analyzing x86 Executables. In *Proceedings of Conference on Compiler Construction*, volume 3443 of *LNCS*, pages 250–254. Springer, 2005.
- [4] T. Ball and S.K. Rajamani. The SLAM Toolkit. In *Computer aided verification*, volume 2102 of *LNCS*, pages 260 – 264. International Conference on Computer-Aided Verification 13, 2001, Paris; CAV-01, Springer, 2001.
- [5] T. Ball and S.K. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Program Analysis*, pages 1–3. ACM Press, 2002.
- [6] F. Bourdoncle. Abstract Interpretation By Dynamic Partitioning. *Journal of Functional Programming*, 1993.
- [7] F. Bourdoncle. Abstract Debugging of Higher-Order Imperative Languages. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 46–55. ACM Press, 1993.
- [8] E. Clarke and D. Kroening. ANSI-C Bounded Model Checker - User Manual. Technical report, Carnegie Mellon University, 2003.
- [9] P. Cousot and R. Cousot. Static Determination of Dynamic Properties of Programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.

- [10] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth Annual Symposium on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.
- [11] P. Cousot and R. Cousot. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In *Proceedings of the International Workshop Programming Language Implementation and Logic Programming*, LNCS 631, pages 269–295. Springer-Verlag, Berlin, Germany, 1992.
- [12] P. Cousot and R. Cousot. Abstract Interpretation Frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.
- [13] D.R. Engler and M. Musuvathi. Static Analysis Versus Software Model Checking for Bug Finding. In *Proceedings of the 5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI-2004)*, volume 2937 of LNCS, pages 191–210. Springer, 2004.
- [14] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Program Analysis*, pages 58–70, 2002.
- [15] M. Musuvathi, D.Y.W. Park, A. Chou, D.R. Engler, and D.L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. In *Proceedings of the 5th ACM Symposium on Operating System Design and Implementation (OSDI-02)*, Operating Systems Review, pages 75–88. ACM Press, 2002.
- [16] G. Necula, S. McPeak, and W. Weimer. CCured: Type-Safe Retrofitting of Legacy Code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Program Analysis*, pages 128–139, 2002.
- [17] G. Necula, S. McPeak, S.P. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proceedings of Conference on Compiler Construction*, 2002.
- [18] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.
- [19] T.W. Reps, G. Balakrishnan, and J Lim. Intermediate-Representation Recovery from Low-Level Code. In *Proceedings of Workshop on Partial Evaluation and Program Manipulation (PEPM)*, 2006.
- [20] B. Blanchet. Introduction to Abstract Interpretation. *Lecture Notes*, 2003.
- [21] BLAST, January 2006. <http://embedded.eecs.berkeley.edu/blast/>.
- [22] F. Bourdoncle, January 2006. <http://www.exalead.com/Francois.Bourdoncle/>.
- [23] G. Necula. CIL, January 2006. <http://www.cs.berkeley.edu/~necula/>.
- [24] SLAM Project, January 2006. <http://research.microsoft.com/slam/>.