Generic Value-Set Analysis on Low-Level Code

Zhenkai Zhang   Xenofon Koutsoukos

EECS Department, Vanderbilt University, USA

{zhenkai.zhang,xenofon.koutsoukos}@vanderbilt.edu

Abstract

Verification is an essential task in the design of cyber-physical systems (CPS). Due to the lack of many details captured at a high-level like compilation and computational platform limitations, CPS software also needs to be analyzed/verified at a low-level. Value-set analysis (VSA) has been proposed to perform both numeric and pointer analyses on low-level code, but it is originally developed for Intel x86 architecture. If we want to perform VSA for a new architecture, we need to make changes to the analysis programs taking into account the semantics of its instructions, which can be tedious and error-prone. In this paper, we address this challenge by using an intermediate language to capture the semantics of the instructions of different architectures. Then, we define the abstract semantics with respect to VSA for the intermediate language. We give an example to show the feasibility of the generic VSA approach, which is to precisely resolve indirect branch target addresses. In addition, we also extend the original strided-interval domain to enable more precise tracking of structured values, and define the operations on this extended strided-interval domain.

Keywords Static Analysis, Verification

1. Introduction

Cyber-physical systems (CPS) are complex systems that are characterized by tight interactions between the physical dynamics, computational platforms, communication networks, and control software. Many CPS are safety-critical or mission-critical systems, such as aerial vehicles and defense missiles, which means any failure may cause a great damage. Since buggy software is notoriously responsible for many system failures, it is an essential task to analyze/verify various properties of CPS software to guarantee it conforms to the specification.

When analyzing/verifying many CPS software properties, it may be insufficient to only consider this software in its high-level form, e.g. its source code. Often, we also need to analyze/verify these properties in its low-level form, e.g. its machine code, namely we need to take into account the compilation that transforms the source code into the low-level code and the interactions between the compiled software and its underlying computational platform; otherwise, even the high-level code is verified, compiler-induced bugs or unexpected architectural limitations may still cause the system to fail. For example, it is reported in [13] that every tested compiler is found to be able to generate wrong code silently; and a Patriot missile failed to intercept a Scud missile in the Gulf War due to the precision error of time calculation using a 24-bit fixed point register.

Value-set analysis (VSA) has been proposed to simultaneously perform numeric and pointer analyses on low-level code [2], which can be used to analyze/verify various control software properties (e.g. variable range and saturation) and security vulnerabilities (e.g. buffer overflow and side channels) for a specific platform. However, the original work only targets at the Intel x86 instruction set architecture (ISA). Thus, if we want to perform VSA on binaries in other ISAs, we need to make changes repeatedly to take into account the semantics of their instructions, which can be a tedious and error-prone process.

While there is some uniqueness in different ISAs, many instructions of an ISA have their counterparts in another ISA, and they share a lot of similarities in their semantics. Thus, if we can use an intermediate language to encode the instructions of different ISAs capturing their computational semantics, we can use one generic VSA program which analyzes the translated binaries in this intermediate language, instead of modifying the VSA programs to target at different ISAs.

The main contributions of this paper are: (1) we extend the original strided-interval domain in order to prevent huge over-approximation from happening in the case of wrap-around computations; (2) we define several operations on the extended strided-interval domain which can more precisely track the set of structured numbers; (3) we use an Intermediate Static Analysis Language (iSAL) with a straightforward concrete semantics to encode instructions of different ISAs; (4) we define an abstract semantics for the iSAL in the value-set abstract domain to facilitate any VSA program writing; (5) we give an example to show the feasibility of the generic VSA approach, which is to precisely resolve indirect branch target addresses.

The 5th Analytic Virtual Integration of Cyber-Physical Systems Workshop
December 2, 2014, Rome, Italy.

AVICPS website:
http://www.analyticintegration.org/

The rest of the paper is organized as: Section 2 states the related work; Section 3 briefly describes VSA and presents the extension to the original strided-interval domain with a set of operations; Section 4 introduces the syntax and concrete semantics of iSAL and defines abstract semantics with respect to VSA; Section 5 discusses some issues when using iSAL to perform VSA; Section 6 presents an example on resolving indirect branches and Section 7 concludes this work.

2. Related Work

In order to realize generic analysis on low-level code of different architectures, semantically translating the low-level code to a generic intermediate form becomes necessary. Intermediate languages are actually often used in many compiler frameworks, like GCC and LLVM, to enable the support of different programming languages at the front end and of different platforms at the back end.

In order to make it possible to develop analysis tools and algorithms for generic security analysis on binary code, an intermediate language called REIL is proposed in [7]. REIL has a very compact set of intermediate instructions while being able to achieve semantic closure, but the translated code in REIL may have a very long representation. Later in [11], an extension to REIL called RREIL is made mainly with an addition of several comparison instructions to take into account the relational information. Based on RREIL, a toolkit called GDSL to specify semantics to machine languages is presented in [8].

In [5], a binary analysis platform (BAP) is introduced which is a successor of Vine that is used in the BitBlaze project [12]. Both BAP and Vine use a formally specified intermediate language to enable generic static analysis. In [3], a binary code analysis framework called BINCOA is proposed that is based on a formal automaton model to capture the low-level code semantics.

The original VSA is summarized in [2], and is used in a tool called CodeSurfer/x86 for analyzing Intel x86 binary code [1]. In order to achieve adaptable value analysis, intermediate representations are also used in [4]. Incremental SAT solving is used to derive sets of values for registers taking into account the relationship between registers and status flags.

An approach that shares a lot of similarities with ours is described in [10], in which the authors introduce a numerical abstract domain called *CLP* (circular linear progression). Their *CLP* serves exactly the same purpose as our *ESI* (extended strided-interval) domain, which is used to precisely track a set of structured integers. Different from their work, we focus on an intermediate language whose abstract semantics is specified upon an abstract domain that is constructed from *ESI*.

3. Value-Set Analysis

In this section, we first briefly recapitulate VSA, and then we argue why it is necessary to extend the original strided-interval domain, and redefine several operations on the extended domain.

VSA combines numeric analysis and pointer analysis together, and its goal is to determine an over-approximation of the set of numeric values and addresses at each program point [2]. It is a flow-sensitive, context-sensitive, and interprocedural static analysis approach based on abstract interpretation [6].

In order to avoid dependence on absolute memory addresses (since some of them may not be determined statically), the memory model in VSA consists of a set of separated memory regions, and each memory region is an abstract memory space which corresponds to the set of all concrete memory spaces with respect to specific run-time properties, e.g. all the possible stack frames of a procedure invocation are aggregated as a memory region. There are three types of memory regions in VSA: the global-region for statically allocated variables in the program, local-regions for locally allocated variables in the run-time stack, and heap-regions for dynamically allocated variables in the heap. While there is only one global-region, there are as many local-regions as procedures.

An abstract address in a memory region can be represented by a pair $\langle \textit{memory-region}, \textit{value} \rangle$, which corresponds to the set of all memory addresses a variable can have. A global variable has a fixed address which can be represented by $\langle \textit{global-region}, \textit{address} \rangle$, while a local variable has a varying address but a fixed offset to the varying base address of the stack frame and its abstract address can be represented by $\langle \textit{local-region}, \textit{offset} \rangle$.

The explicitly referenced variables in the source code are accessed by using their addresses in the machine code, VSA needs to recover the variables from low-level code, and represent them by using abstract locations (*a-locs*). An *a-loc* is a variable-like entity which may have an explicit boundary (e.g. registers) or an implicit boundary (e.g. variables allocated in the memory).

Given an *a-loc*, the abstract state of VSA maps it to a value-set. A value-set is a function that maps a memory region to a strided-interval (*SI*). A strided-interval is an abstract object used to represent a set of structured integers with a fixed stride. A k-bit strided-interval $s[l, u]$ where $-2^{k-1} \leq l \leq u \leq 2^{k-1} - 1$, $0 \leq s \leq 2^k - 1$ represents the set $\{i | i = l + n \times s \wedge n \geq 0 \wedge i \leq u\}$. Depending on the type of the given memory region, the mapped strided-interval has different semantics: if the memory region is the global-region, the mapped strided-interval represent a set of numeric values which are the values held by the *a-loc* in some executions; otherwise, the mapped strided-interval represent a set of offsets in the memory region, and each $\langle \textit{memory-region}, \textit{offset} \rangle$ pair is an abstract address with respect to the memory region. For a given memory region, the mapped strided-interval may be a $\bot$ which means the empty set of values. If the mapped strided-interval is a $\top$, it is the set of all the representable values $1[-2^{k-1}, 2^{k-1}-1]$.

3.1 Extension to Strided-Interval

The biggest problem of the original strided-interval representation is that: if l is required to be less than or equal to u (i.e. $\forall s[l, u], l \leq u$), it loses the ability to precisely track the set of numbers when some of the numbers in the set lead to an overflow interpretation with respect to two's complement representation. For example, on a 16-bit architecture, if a strided-interval **4[0x7FF0, 0x7FFC]** is added another interval **0[4, 4]**, according to the addition operation defined in [9], the resultant strided-interval will be **1[0x8000, 0x7FFF]**, i.e. the $\top$ element in the *SI* domain. Although the result is sound, basically it treats the upper bound of the calculation as an overflow and let $\top$ safely capture all possible values, it is not precise at all.

Actually, in the example the interval **4[0x7FF4, 0x8000]** would be a more precise and also sound result, in which l represents a signed positive number $2^{15} - 12$ and u represents a signed negative number -2^{15}. Thus, a better decision is to get rid of the constraint $l \leq u$ so as to allow both l and u to be any number of the range $[-2^{k-1}, 2^{k-1} - 1]$, which induces our extended strided-interval (*ESI*) domain similar to the *CLP* domain in [10]. Let γ denote the concretization function mapping an extended strided-interval $s[l, u] \in ESI$ to a set of integers, and we have

$$\gamma(s[l,u]) = \begin{cases} \{i | i = l + n \times s \wedge n \geq 0 \wedge i \leq u\} & if\ l \leq u \\ \{i | i = l + n \times s \wedge n \geq 0 \wedge i \leq u + 2^k\} & otherwise \end{cases}$$

3.2 Operations on Extended Strided-Interval

There are six groups of operations on *ESI*, which are arithmetic, shift, bit-wise, set, comparison, and truncation operations.

Let us assume the underlying platform is a n-bit architecture, and let max_u be the number $2^n - 1$, max_s be the number $2^{n-1} - 1$, and min_s be the number -2^{n-1}. For an arbitrary number x, $n(x)$ gives the lowest n bits representation of x, and $tz(x)$ gives the number of trailing zeros in x's representation.

In addition, let us define several functions that are used in the operations: let $sg(p, q, s)$ return the *smallest* number that is *greater* than p and can be reached by q in multiple s strides, and let $gs(p, q, s)$ return the *greatest* number that is *smaller* than p and can be reached by q in multiple s strides (the computation wraps around in terms of n-bit).

Let $dsi_u(s[l, u])$ give an ordered set of disjoint strided-intervals, each of which represents a maximal sub-interval with respect to unsigned integers. Depending on the $s[l, u]$, this set can be a singleton (e.g. $dsi_u(1[10, 20]) = \{1[10, 20]\}$), or has two members (e.g. $dsi_u(1[-2, 2]) = \{1[0, 2], 1[max_u - 1, max_u]\}$). Let $fst(dsi_u(s[l, u]))$ give the first member in this ordered set, and let $snd(dsi_u(s[l, u]))$ give the second one if it exists, or $\perp$ if the set is a singleton. Similarly, let $dsi_s(s[l, u])$ give an ordered set of disjoint strided-intervals, each of which represents a maximal sub-interval with respect to signed integers.

Arithmetic Operations

For an **addition** operation $s_1[l_1, u_1] +^{si} s_2[l_2, u_2]$, let us assume $s = \gcd(s_1, s_2)$, $\hat{l} = l_1 + l_2$, and $\hat{u} = u_1 + u_2$ without overflow, i.e. $\hat{l}$ and $\hat{u}$ have enough bits to contain the sums. From now on, let us assume all the arithmetic operations on numeric values will not induce overflow.

$$s_1[l_1, u_1] +^{si} s_2[l_2, u_2] = \begin{cases} s[n(\hat{l}), n(\hat{u})] & if\ \hat{u} - \hat{l} < 2^n \\ s[sg(min_s, \hat{l}, s), gs(max_s, \hat{u}, s)] & else\ if\ s = 2^m \\ 1[min_s, max_s] & otherwise \end{cases}$$

For a **negation** operation $-^{si}s_1[l_1, u_1]$, since *ESI* allows $l_1 > u_1$, we can easily have

$$-^{si}s_1[l_1, u_1] = s_1[-u_1, -l_1]$$

which is simpler but more precise than the negation operation for the original strided-interval defined in [9]. Thus, for a **subtraction** operation $s_1[l_1, u_1] -^{si} s_2[l_2, u_2]$, we have

$$s_1[l_1, u_1] -^{si} s_2[l_2, u_2] = s_1[l_1, u_1] +^{si} (-^{si}s_2[l_2, u_2])$$

For an **unsigned multiplication** operation $s_1[l_1, u_1] \times_u^{si} s_2[l_2, u_2]$, we have $prod_u$ defined as

$$prod_u = \{p | \exists s_x[l_x, u_x] \in dsi_u(s_1[l_1, u_1]), \\ s_y[l_y, u_y] \in dsi_u(s_2[l_2, u_2]) : p = l_x \times l_y \vee p = u_x \times u_y\}$$

Let $\hat{l}_1$ be the lower bound of $fst(dsi_u(s_1[l_1, u_1]))$ and let $\hat{l}_2$ be the lower bound of $fst(dsi_u(s_2[l_2, u_2]))$. We have

$$\hat{s}_1 = \gcd(s_1 \times s_2, \hat{l}_1 \times s_2, \hat{l}_2 \times s_1)$$
$$\hat{s}_2 = \gcd(\hat{s}_1, \hat{l}_1 \times 2^n, s_1 \times 2^n)$$
$$\hat{s}_3 = \gcd(\hat{s}_1, \hat{l}_2 \times 2^n, s_2 \times 2^n)$$
$$\hat{s}_4 = 2^{tz(\gcd(\hat{s}_2, \hat{s}_3))}$$

$$\hat{s} = \begin{cases} \hat{s}_1 & if\ |dsi_u(s_1[l_1, u_1])| = 1 \wedge |dsi_u(s_2[l_2, u_2])| = 1 \\ \hat{s}_2 & if\ |dsi_u(s_1[l_1, u_1])| = 1 \wedge |dsi_u(s_2[l_2, u_2])| = 2 \\ \hat{s}_3 & if\ |dsi_u(s_1[l_1, u_1])| = 2 \wedge |dsi_u(s_2[l_2, u_2])| = 1 \\ \hat{s}_4 & if\ |dsi_u(s_1[l_1, u_1])| = 2 \wedge |dsi_u(s_2[l_2, u_2])| = 2 \end{cases}$$

Since the product of two n-bit numbers should be contained in $2n$-bit, we have the multiplication operation to generate two *ESI*s: the first *ESI* corresponds to the high n-bit of the product, and the second one corresponds to the low n-bit of the product. Let $p_{min} = \min(prod_u)$, and $p_{max} = \max(prod_u)$. We have

$$s_1[l_1, u_1] \times_u^{si} s_2[l_2, u_2] = \langle 1[\lfloor \frac{p_{min}}{2^n} \rfloor, \lfloor \frac{p_{max}}{2^n} \rfloor], \hat{s}[\hat{l}, \hat{u}] \rangle$$
$$\text{where } [\hat{l}, \hat{u}] =$$
$$\begin{cases} [n(p_{min}), n(p_{max})] & if\ p_{max} - p_{min} \leq max_u \\ [sg(0, 2^{tz(\hat{s})}, \hat{s}), max_u] & otherwise \end{cases}$$

The **signed multiplication** operation $s_1[l_1, u_1] \times_s^{si} s_2[l_2, u_2]$ is similar but makes use of dsi_s instead of dsi_u.

For an **unsigned division** operation $s_1[l_1, u_1] \div_u^{si} s_2[l_2, u_2]$, we have $quot_u$ defined as

$$quot_u = \{q | \exists s_x[l_x, u_x] \in dsi_u(s_1[l_1, u_1]), \\ s_y[l_y, u_y] \in dsi_u(s_2[l_2, u_2]) : q = \lfloor \frac{l_x}{u_y} \rfloor \vee q = \lfloor \frac{u_x}{l_y} \rfloor\}$$

Let $\hat{l}_2$ be the lower bound of $fst(dsi_u(s_2[l_2, u_2]))$, and let $\hat{q} = \lfloor \frac{s_1}{\hat{l}_2} \rfloor$ and $\hat{r} = \frac{s_1}{\hat{l}_2} - \hat{q}$. We have

$$\hat{s} = \begin{cases} \hat{q} & if\ |\gamma(s_1[l_1, u_1])| = 1 \wedge \hat{r} = 0 \wedge \\ & \quad |dsi_u(s_1[l_1, u_1])| = 1 \wedge \hat{q} = 2^m \\ 1 & otherwise \end{cases}$$

$$s_1[l_1, u_1] \div_u^{si} s_2[l_2, u_2] = \hat{s}[\min(quot_u), \max(quot_u)]$$

For an **signed division** operation $s_1[l_1, u_1] \div_s^{si} s_2[l_2, u_2]$, it is similar but makes use of dsi_s instead of dsi_u, and the corresponding $quot_s$ also contains q that is either $q = \lfloor \frac{l_x}{l_y} \rfloor$ or $q = \lfloor \frac{u_x}{u_y} \rfloor$.

Shift Operations

Since a shift operation on a value (left or right, logical or arithmetic, but not circular) makes no difference when the numbers of bits to shift are greater than n, we define $shn(s[l, u])$ as

$$shn(s[l, u]) = \{x | 0 \le x \le n \wedge x \in \gamma(s[l, u])\}$$

to give the set of numbers by which the shift operations are performed usefully. For a logical/arithmetic **left-shift** operation $s_1[l_1, u_1] \ll^{si} s_2[l_2, u_2]$, we extract the useful sub-interval from $s_2[l_2, u_2]$ for the operation:

$$x_{min} = \min(shn(s_2[l_2, u_2]))$$
$$x_{max} = \max(shn(s_2[l_2, u_2]))$$
$$\hat{s}_2[\hat{l}_2, \hat{u}_2] = (2^{x_{min}} \times (2^{s_2} - 1))[2^{x_{min}}, 2^{x_{max}}]$$
$$s_1[l_1, u_1] \ll^{si} s_2[l_2, u_2] = snd(s_1[l_1, u_1] \times_s^{si} \hat{s}_2[\hat{l}_2, \hat{u}_2])$$

For a **logical right-shift** operation $s_1[l_1, u_1] \gg_l^{si} s_2[l_2, u_2]$, it is similar but the result is the quotient of $s_1[l_1, u_1] \div_u^{si} \hat{s}_2[\hat{l}_2, \hat{u}_2]$.

For an **arithmetic right-shift** operation $s_1[l_1, u_1] \gg_a^{si} s_2[l_2, u_2]$, it is also similar but the result is the quotient of $s_1[l_1, u_1] \div_s^{si} \hat{s}_2[\hat{l}_2, \hat{u}_2]$. Different from logically shifting, an arithmetic right-shift operation needs to fill in the sign bit of the shifted number. In the case of $|shn(s_2[l_2, u_2])| = 0 \wedge |\gamma(s_2[l_2, u_2])| \neq 0$, it means every number in $\gamma(s_2[l_2, u_2])$ is greater than n. Thus, we also have

$$if\ |shn(s_2[l_2, u_2])| = 0 \wedge |\gamma(s_2[l_2, u_2])| \neq 0$$
$$s_1[l_1, u_1] \gg_a^{si} s_2[l_2, u_2] =$$
$$\begin{cases} 0[-1, -1] & if\ \forall y \in \gamma(s_1[l_1, u_1]) : y < 0 \\ 0[0, 0] & if\ \forall y \in \gamma(s_1[l_1, u_1]) : y \ge 0 \\ 1[-1, 0] & otherwise \end{cases}$$

Bit-Wise Operations

Since the **bit-wise not** operation $\sim^{si} s_1[l_1, u_1]$, the **bit-wise or** operation $s_1[l_1, u_1]\ |^{si} s_2[l_2, u_2]$, the **bit-wise and** $s_1[l_1, u_1]\ \&^{si} s_2[l_2, u_2]$, and the **bit-wise xor** operation $s_1[l_1, u_1] \oplus^{si} s_2[l_2, u_2]$ are similar to the corresponding one defined in [9], we will not state them here.

Set Operations

For a **set union** operation $s_1[l_1, u_1] \cup s_2[l_2, u_2]$, let $\hat{s} = \gcd(s_1, s_2)$, and let us define four boolean variables: $b_1 = l_2 \in \gamma(\hat{s}[l_1, u_1])$, $b_2 = u_2 \in \gamma(\hat{s}[l_1, u_1])$, $b_3 = l_1 \in \gamma(\hat{s}[l_2, u_2])$, and $b_4 = u_1 \in \gamma(\hat{s}[l_2, u_2])$. We have

$$s_1[l_1, u_1] \cup s_2[l_2, u_2] = \begin{cases} \hat{s}[l_a, u_a] & if\ b_1 \wedge b_2 \wedge b_3 \wedge b_4 \\ \hat{s}[l_1, u_1] & else\ if\ b_1 \wedge b_2 \\ \hat{s}[l_2, u_2] & else\ if\ b_3 \wedge b_4 \\ \hat{s}[l_1, u_2] & else\ if\ b_1 \wedge b_4 \\ \hat{s}[l_2, u_1] & else\ if\ b_2 \wedge b_3 \\ \hat{s}_b[l_b, u_b] & otherwise \end{cases}$$

where $[l_a, u_a] =$

$$\begin{cases} [l_1, u_1] & if\ l_1 = l_2 \wedge u_1 = u_2 \\ [sg(min_s, l_1, \hat{s}), gs(max_s, u_1, \hat{s})] & otherwise \end{cases}$$

and the computation of $\hat{s}_b[l_b, u_b]$ is similar to the set union operation given in [10]. The **set intersection** operation $s_1[l_1, u_1] \cap^{si} s_2[l_2, u_2]$ and the **set complement** operation $s_1[l_1, u_1] \backslash^{si} s_2[l_2, u_2]$ are similar to the ones defined in [10] and are not described due to the space limitation.

Comparison and Truncation Operations

For a **comparison** operation $s_1[l_1, u_1] \mathscr{R} s_2[l_2, u_2]$, where $\mathscr{R}$ is a relational operator, we compare the ranges of the members of $dsi_{u|s}(s_1[l_1, u_1])$ and $dsi_{u|s}(s_2[l_2, u_2])$. If the operation is to compare unsigned numbers, we use dsi_u; otherwise we use dsi_s.

For a **truncation** operation $s_1[l_1, u_1] \downarrow^{si} s_2[l_2, u_2]$, we assume $s_2[l_2, u_2]$ give the set of numbers of bits kept in the truncated value. In order to be meaningful, the number of kept bits given by $s_2[l_2, u_2]$ should be smaller than n; otherwise the original number will not be truncated. We borrow $shn(s_2[l_2, u_2])$ from the shift operations defined above. Given a number $x < n$, let us define $trun(s[l, u], x)$ as

$$trun(s[l, u], x) =$$
$$\begin{cases} 0[l \& (2^x - 1), u \& (2^x - 1)] & if\ l = u \\ s[l, u] & if\ \forall y \in \gamma(s[l, u]) : y \& 2^x = 0 \\ 1[0, 2^x - 1] & otherwise \end{cases}$$

and we have

$$s_1[l_1, u_1] \downarrow^{si} s_2[l_2, u_2] = \bigcup_{x \in shn(s_2[l_2, u_2])} trun(s_1[l_1, u_1], x)$$

4. Intermediate Static Analysis Language (iSAL) with Value-Set Analysis Semantics

The iSAL consists of 25 intermediate instructions which are used to encode the semantics of instructions of different ISAs. These instructions are selected between trade-offs in expressivity and compactness (namely, some instructions may be redundant since they can be represented by a combination of others, but their presence makes the translation much easier).

4.1 Syntax and Concrete Semantics

From Tab. 1, we can observe that most of the intermediate instructions have three operands (only two of them have two operands, i.e. **not** and **brc**). In the table, we use *r* to restrict the operand to be a register, and use *f* to represent the operand is a status flag. There are no restrictions on *s* and *t*, namely, each of them can be either a register or an immediate number.

The first 10 instructions are arithmetic instructions. For an arithmetic operation $*$, let $*^m$ denote the result of this operation is in m-bit. Therefore, if the result needs more than m bits to represent, there is a potential overflow. Two functions, hi and lo, are defined as using the high $\frac{m}{2}$ bits and the low $\frac{m}{2}$ bits of a m-bit number respectively. Furthermore, we use $*_u$ to denote the operation treat the operands as unsigned numbers and use $*_s$ to denote the operation treat the operands as signed numbers.

The next 3 instructions are about shift operations. For the left-shift operation, $\lll_0^n$ means the result is confined in n bits (discarding the bits higher than n) and 0 is shifted in from the right to the left, namely, the operation is the logical left-shift. For the right-shift operations, $\ggg_0$ is the logical right-shift operation which shifts 0 in from the left to the right and $\ggg_{msb(s)}$ is the arithmetic right-shift operation which shifts the sign bit of *s* in (i.e. the most significant bit of *s* given by the function msb).

We have 4 instructions for bit-wise operations, although we can just include **not** and **or** instructions and deduce **and** and **xor** instructions by De Morgan laws. Thus, there is a trade-off between compactness and expressivity.

The next 5 comparison instructions are used to check the relations between two operands. As the arithmetic instructions, they distinguish between signed and unsigned comparisons. If the designated relation is met, the first status flag operand will be set; otherwise, the flag will be cleared.

ld and **st** are the only two instructions to operate memory. The second operand *s* gives the load/store size in bytes and the third operand *t* gives the base address of the memory operation. Given an address range, the mem function returns the corresponding collection of memory cells.

The sequential control flow can only be changed by the conditional branch instruction, i.e. **brc** instruction. The unconditional branches instructions can be modeled by setting the first status flag operand as always-set.

Translation from a binary executable $\mathcal{B}$ in some ISA into the corresponding program $\mathcal{B}^T$ in iSAL can be achieved automatically provided the mapping of instructions is available. The encoded mapping captures the semantics of the instructions of the ISA using the iSAL.

The concrete semantics of a binary executable program (and its translated intermediate program) considers every possible execution path in all possible environments. It may be an infinite mathematical object which is not computable. In order to make the analysis tractable, some form of over-approximation is needed. Abstract interpretation [6] is proposed to formalize the notion of over-approximation in a unified framework. Based on abstract interpretation, an analysis like VSA can be used to verify the properties of CPS software.

Table 1. Syntax and Concrete Semantics of 25 Intermediate Instructions

Instruction	Concrete Semantics
add *r*, *s*, *t*	$r := s +^n t$
sub *r*, *s*, *t*	$r := s -^n t$
muluhi *r*, *s*, *t*	$r := hi(s \times_u^{2n} t)$
mululo *r*, *s*, *t*	$r := lo(s \times_u^{2n} t)$
mulshi *r*, *s*, *t*	$r := hi(s \times_s^{2n} t)$
mulslo *r*, *s*, *t*	$r := lo(s \times_s^{2n} t)$
divu *r*, *s*, *t*	$r := s \div_u t$
divs *r*, *s*, *t*	$r := s \div_s t$
modu *r*, *s*, *t*	$r := s \bmod_u t$
mods *r*, *s*, *t*	$r := s \bmod_s t$
shl *r*, *s*, *t*	$r := s \lll_0^n t$
shrl *r*, *s*, *t*	$r := s \ggg_0 t$
shra *r*, *s*, *t*	$r := s \ggg_{msb(s)} t$
and *r*, *s*, *t*	$r := s \,\&\, t$
or *r*, *s*, *t*	$r := s \mid t$
not *r*, *s*	$r := \sim s$
xor *r*, *s*, *t*	$r := s \oplus t$
cmpeq *f*, *s*, *t*	if $s = t$ then $set(f)$ else $clr(f)$
cmpleu *f*, *s*, *t*	if $s \leq_u t$ then $set(f)$ else $clr(f)$
cmples *f*, *s*, *t*	if $s \leq_s t$ then $set(f)$ else $clr(f)$
cmpltu *f*, *s*, *t*	if $s <_u t$ then $set(f)$ else $clr(f)$
cmplts *f*, *s*, *t*	if $s <_s t$ then $set(f)$ else $clr(f)$
ld *r*, *s*, *t*	$r := mem(t, t+s)$
st *r*, *s*, *t*	$mem(t, t+s) := r$
brc *f*, *t*	if $isset(f)$ then $goto(t)$

4.1.1 Abstract Domains for VSA

Following the original work of VSA (which is summarized in [2] and has been briefly described in Section 3), we define several abstract domains that are used in VSA.

In our virtual iSAL architecture, in addition to the registers in the encoded ISA, temporary registers can be declared and used in order to keep intermediate values in the process of an instruction execution. There can be as many temporary registers as needed. Let $NormLoc$ denote the set of ordinary *a-locs* corresponding to target ISA registers, global variables, and local variables, let $FlagLoc$ denote the set of *a-locs* corresponding to status flags used in the encoding, and let $TempLoc$ denote the set of temporary *a-locs* corresponding to the declared temporary registers and other entities which hold temporary values. We have $AbsLoc$ defined as

$$AbsLoc = NormLoc \cup FlagLoc \cup TempLoc$$

Let $MemRgn$ denote the set of all the memory regions, which include the single global-region and all the local-regions (since CPS software seldom use dynamic memory allocation, usually we can ignore heap-regions), and let VS denote the set of all the value-sets. Thus, we have

$$VS = MemRgn \rightarrow ESI_{\perp}$$

where $ESI_{\perp}$ is the lifted extended strided-interval domain, i.e. $ESI_{\perp} = ESI \cup \{\perp\}$. Thus, there is a special value-set $vs_{\perp} \in VS$ such that $\forall mr \in MemRgn : [mr \mapsto \perp]$. We also have another special value-set $vs_{\top} \in VS$ such that $\forall mr \in MemRgn : [mr \mapsto \top]$.

Let B^3 denote the Kleene three-valued logic domain, i.e. $B^3 = \{\mathit{TRUE}, \mathit{FALSE}, \mathit{UNKOWN}\}$. Let gr denote the global-region. Given a $b \in B^3$, we define an auxiliary function $bvs : B^3 \to VS$ as

$$bvs(b) = \begin{cases} vs_{\perp}[gr \mapsto 0[1,1]] & if\ b = \mathit{TRUE} \\ vs_{\perp}[gr \mapsto 0[0,0]] & if\ b = \mathit{FALSE}\ ^{1} \\ vs_{\perp}[gr \mapsto 1[0,1]] & otherwise \end{cases}$$

Also, let us define $vsb : VS \to B^3$ as the inverse operation of bvs. In order to facilitate specifying the abstract semantics for the comparison instructions, we define a product domain FS as

$$FS = VS \times AbsLoc \times VS \times VS$$

Given a $fs \in FS$, the first component of fs (denoted as $fs\langle 1\rangle$) is the answer of function bvs applied to the result of a comparison, the second component ($fs\langle 2\rangle$) is the *a-loc* of the second operand in a comparison instruction, the third component ($fs\langle 3\rangle$) is the partition of the value-set mapped from $fs\langle 2\rangle$ that makes the comparison TRUE, and the last component ($fs\langle 4\rangle$) is the partition that makes the comparison FALSE.

An abstract state of VSA maps an *a-loc* $a \in AbsLoc$ to a $vs \in VS$ if $a \in NormLoc \cup TempLoc$, or to a $fs \in FS$ if $a \in FlagLoc$. Let $State$ denote the set of all the abstract states of VSA. We have

$$State = AbsLoc \to VS \cup FS$$

Since the translation for a given binary executable program is has a finite length and the target architecture has a fixed word size, all the domains described above are finite.

4.1.2 Abstract Semantics for VSA

Each intermediate instruction in the iSAL has an abstract semantic function for VSA which transforms an abstract state to another state(s). Let $IInst$ denote the set of all the 25 intermediate instructions. Formally, we have this semantic mapping function

$$sm : IInst \to (State \to State^{+})$$

which assigns each intermediate instruction an abstract semantic function on $State$. Let us also define an auxiliary function

$$al : IInst \times \{1,2,3\} \to AbsLoc$$

that maps the i^{th} operand of an intermediate instruction to its corresponding *a-loc*. For a two-operand instruction χ, i.e. **not** or **brc** instruction, let $al(\chi, 3)$ give $\epsilon \in TempLoc$ such that $\forall \sigma \in State : \sigma(\epsilon) = vs_{\perp}$.

[1] Given a function $f : A \to B$, let $f[x \mapsto y]$ mean $f(x) = y$ and $\forall a \in A \wedge a \neq x : f(a) = f(a)$

Let $ASB \subset IInst$ denote the set of intermediate instructions in the first three groups (i.e. the arithmetic, shift, and bit-wise instructions). Given an instruction $\alpha \in ASB$ and an abstract state $\sigma \in State$, we have

$$sm(\alpha)(\sigma) =$$
$$\begin{cases} \sigma[al(\alpha,1) \mapsto op(\alpha)^{vs}\sigma(al(\alpha,2))] & if\ al(\alpha,3) = \epsilon \\ \sigma[al(\alpha,1) \mapsto \sigma(al(\alpha,2))op(\alpha)^{vs}\sigma(al(\alpha,3))] & otherwise \end{cases}$$

where $op(\alpha)$ gives the corresponding operation the instruction semantically performing, and $op(\alpha)^{vs}$ denotes the operation is performed on VS domain. The operations on VS domain are based on the operations on strided-interval domain (ESI domain in our case), which are defined in [9].

Let $CMP \subset IInst$ denote the set of comparison instructions. Given an instruction $\beta \in CMP$ and an abstract state $\sigma \in State$, we have

$$\begin{gathered} sm(\beta)(\sigma) = \sigma[al(\beta,1) \mapsto fs] \text{ where} \\ fs\langle 1\rangle = bvs(al(\beta,2)op(\beta)^{vs}al(\beta,3)) \wedge \\ fs\langle 2\rangle = al(\beta,2) \wedge \\ fs\langle 3\rangle(op(\beta)^{vs})^{-1}\sigma(al(\beta,3)) = \mathit{FALSE} \wedge \\ fs\langle 4\rangle op(\beta)^{vs}\sigma(al(\beta,3)) = \mathit{FALSE} \end{gathered}$$

where $(op(\beta)^{vs})^{-1}$ gives the inverse relational operation of $op(\beta)^{vs}$. The reason of using inverse operation is: in the case of the comparison giving $\mathit{TRUE}/\mathit{FALSE}$ instead of $\mathit{UNKNOWN}$, $fs\langle 4\rangle/fs\langle 3\rangle$ is $vs_{\perp}$ and we assume a relational operation on $vs_{\perp}$ always gives FALSE. In terms of comparing two value-sets, we only compare them if they have the same VS type – either VS_{global} (i.e. having all the memory regions mapped to $\perp$ except for the global-region) or VS_{single} with the same valid memory region mr (i.e. having all the memory regions mapped to $\perp$ except for the mr); otherwise, the comparison gives $\mathit{UNKNOWN}$, and both $fs\langle 3\rangle$ and $fs\langle 4\rangle$ are set as $\sigma(al(\beta,2))$.

Let η be either a **ld** or a **st** instruction, which uses the second operand to specify the load/store size. Since there is barely an architecture that has a varying size in a specific load/store instruction, we can assume $vs_2 = \sigma(al(\eta,2)) \in VS_{global}$ and $vs_2(gr) = 0[w,w]$ where $\sigma \in State$ and w is a valid size value that can be loaded/stored in the target architecture. η also uses the third operand to specify the base memory address, which can be an address of a static object, or an address of an object that is allocated in stack. For a $vs \in VS$, let us define a function $rg : VS \to MemRgn$ such that $rg(vs)$ gives the global-region if $vs \in VS_{global}$; otherwise $rg(vs)$ gives the local-region of the procedure that is under analysis. Let $vs_3 = \sigma(al(\eta,3))$, and let $Addr$ be the set of *a-locs* that are constructed from $w, rg(vs_3)$ and $\gamma(vs_3(rg(vs_3)))$. Let $\overleftarrow{\eta}$ be a **ld** instruction. We have

$$sm(\overleftarrow{\eta})(\sigma) = \sigma[al(\overleftarrow{\eta},1) \mapsto \sqcup^{vs}_{d \in Addr}\sigma(d)]$$

where $\sqcup^{vs}$ is the join operation on VS which is to memory region-wisely join extended strided-intervals.

In terms of storing, there may be some *a-locs* overlapping with the *a-loc*(s) being modified by a store instruction.

Let $Ovlp$ be the set of *a-locs* that are overlapping with the *a-loc*(s) being modified by a **st** instruction $\overrightarrow{\eta}$. We have

$$sm(\overrightarrow{\eta})(\sigma) = \sigma \left[\begin{array}{l} \forall d \in Addr : d \mapsto \sigma(al(\overrightarrow{\eta}, 1)), \\ \forall o \in Ovlp : o \mapsto vs_\top \end{array} \right]$$

A **brc** instruction δ denotes the end of the current basic block. It is the only way to change the control flow and fork the current state σ depending on $fs = \sigma(al(\delta, 1))$. We have

$$sm(\delta)(\sigma) =$$
$$\begin{cases} \langle \sigma[fs\langle 2\rangle \mapsto fs\langle 3\rangle], \sigma_\perp\rangle & if\ vsb(fs\langle 1\rangle) = TRUE \\ \langle \sigma_\perp, \sigma[fs\langle 2\rangle \mapsto fs\langle 4\rangle]\rangle & if\ vsb(fs\langle 1\rangle) = FALSE \\ \langle \sigma[fs\langle 2\rangle \mapsto fs\langle 3\rangle], \sigma[fs\langle 2\rangle \mapsto fs\langle 4\rangle]\rangle & otherwise \end{cases}$$

where $\sigma_\perp$ means $\forall a \in NormLoc \cup TempLoc : \sigma(a) = vs_\perp \wedge \forall f \in FlagLoc : \sigma(f) = fs_\perp$. Therefore, a **brc** instruction partitions the σ into two ordered parts: the first true part is for the branch taken execution and the second false part is for the fall-through execution.

5. Value-Set Analysis on Translated Programs

VSA on the translated program in iSAL intends to derive the fixed-points of the abstract states of each program point using iterations. In each iteration, we update the abstract states according to the abstract semantics described above.

5.1 Handling Delay Slots

Several architectures (e.g. MIPS) use delay slots to compensate the performance loss when dealing with conditional branches.

Since the **brc** instruction intends to mean the end of a basic block and partitions the $\sigma \in State$ into a true part and a fall-through part, any intermediate instruction following a **brc** instruction will not change the value-sets of the true partition. However, in the presence of delay slots, this does not conform to the original code's semantics.

Fortunately, the instructions in the delay slots are arranged by the compiler which does not allow any of the instructions in the delay slots to have a dependency with the associated conditional branch. Thus, when we perform VSA on a basic block, if the last few instructions are used as delay slots, we can rearrange the order of the analysis by processing them before the corresponding conditional branch.

5.2 Join Function

A *join* semantic function is needed to combine the incoming abstract states when a basic block has more than one predecessors in the control flow graph (CFG). Given two abstract states $\sigma_1 \in State$ and $\sigma_2 \in State$, we have the join function $join : State \times State \rightarrow State$ defined as

$$join(\sigma_1, \sigma_2) = \left[\begin{array}{l} \forall a \in NormLoc : a \mapsto \sigma_1(a) \sqcup^{vs} \sigma_2(a), \\ \forall b \in TempLoc : b \mapsto vs_\perp, \\ \forall f \in FlagLoc : f \mapsto \sigma_1(f) \sqcup^{fs} \sigma_2(f) \end{array} \right]$$

where $\sqcup^{fs}$ is the join operation on FS domain. Given a $fs_1 \in FS$ and a $fs_2 \in FS$, we define $\sqcup^{fs}$ as

$$fs_1 \sqcup^{fs} fs_2 =$$
$$\begin{cases} \langle bvs(UNKNOWN), \epsilon, vs_\perp, vs_\perp\rangle & if\ fs_1\langle 2\rangle \neq fs_2\langle 2\rangle \\ \langle fs_1\langle 1\rangle \sqcup^{vs} fs_2\langle 1\rangle, fs_1\langle 2\rangle, & otherwise \\ \ fs_1\langle 3\rangle \sqcup^{vs} fs_2\langle 3\rangle, fs_1\langle 4\rangle \sqcup^{vs} fs_2\langle 4\rangle\rangle & \end{cases}$$

When joining two abstract states, we discard the value-set information of temporary *a-locs*, since the information is not used in the new basic block. If a basic block has more than two predecessors, a successive joining is performed, i.e. $join(\sigma_n, join(\ldots join(\sigma_1, \sigma_2)))$. Moreover, if a predecessor ends with a **brc** instruction, depending on whether the new basic block is the target of that **brc** instruction, the true/false part of the resultant states is used.

5.3 Handling Input Dependent Value-Sets

When analyzing a program that has input dependent variables, for the sake of safety, these variables are supposed to be any possible numbers. In the context of VSA, an abstract state maps an *a-loc* corresponding to such an input dependent variable to $vs_\top$. Since the result of an operation on a $vs_\top$ is also a $vs_\top$, the propagation of $vs_\top$ will make the analysis imprecise or even useless.

In order to improve the precision of analysis, we keep track of the operations on $vs_\top$ until a **brc** instruction is met. Since the **brc** instruction partitions the state into two parts depending on some previous comparison, in each part, we use the information discovered by the comparison to refine the $vs_\top$ propagation chain. In the current work, we only keep track of the chains of linear operations so as to reduce the complexity.

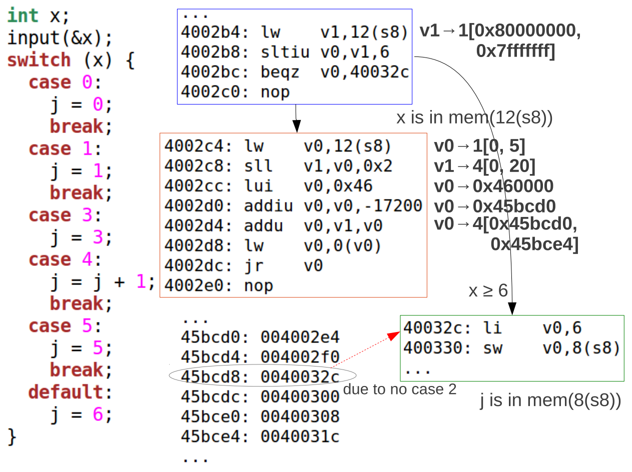
6. Example – Handling Indirect Branches

As model-based control design tools become mature like Simulink, a large part of control software is designed by using formal specifications like finite-state machine (FSM) or state chart, and its C code is generated by using a code generator like Simulink Coder. Usually, the code generator opts for using *switch* statements to implement the transitions between states.

However, the compilers often implement *switch* statements by using indirect branches, whose presence makes reconstructing a whole CFG from a binary very challenging. Since a typically static analysis is performed on the extracted CFG, how to precisely resolve the target addresses of these indirect branches becomes essential.

As an example, we encode most of the instructions of MIPS ISA (without considering floating-point, coprocessor, and exception instructions) using the iSAL, and show VSA can precisely resolve the indirect branch instructions when reconstructing the CFG.

As shown in Fig. 1, the code has an input dependent variable x, and the *switch* statement relies on the value of the input. At the address **0x4002bc**, the **brc** intermediate instruction in the **beqz** MIPS instruction can determine: when v1 ≥ 6, v0 is 0 and the branch will be taken; otherwise, the



```c
int x;
input(&x);
switch (x) {
  case 0:
    j = 0;
    break;
  case 1:
    j = 1;
    break;
  case 3:
    j = 3;
  case 4:
    j = j + 1;
    break;
  case 5:
    j = 5;
    break;
  default:
    j = 6;
}
```

```
...
4002b4: lw    v1,12(s8)
4002b8: sltiu v0,v1,6
4002bc: beqz  v0,40032c
4002c0: nop
```

```
4002c4: lw    v0,12(s8)
4002c8: sll   v1,v0,0x2
4002cc: lui   v0,0x46
4002d0: addiu v0,v0,-17200
4002d4: addu  v0,v1,v0
4002d8: lw    v0,0(v0)
4002dc: jr    v0
4002e0: nop
```

```
...
45bcd0: 004002e4
45bcd4: 004002f0
45bcd8: 0040032c
45bcdc: 00400300
45bce0: 00400308
45bce4: 0040031c
...
```

```
40032c: li    v0,6
400330: sw    v0,8(s8)
...
```

Figure 1. A C Code Snippet with *switch* Statements and Compiled MIPS Code

control falls through. Thus, the partitioned true part of the state has v1 $\geq$ 6 and the false part has $0 \leq$ v1 ≤ 5. However, since the value of x is stored in a local variable whose address is given as (s8+12) in the binary code where s8 has already been set equal to sp (stack pointer register), we also need to handle the value-set mapped from the *a-loc* corresponding to (s8+12) in the true and false parts partitioned by **brc**, whereas the value of v0 at the address **0x4002c4** will still be any possible number as x.

From v0 at the address **0x4002d4**, we can observe that it can have 6 values ranging from **0x45bcd0** to **0x45bce4**, each of which is separated by 4. These 6 values are exact data addresses where the indirect branch target addresses are stored. However, since the target addresses stored at these 6 data addresses are not regularly structured, after the **lw** instruction at the address **0x4002d8**, v0 will have an extended strided-interval **4[0x4002e4, 0x40032c]** which corresponds to 19 values. In order to remove this imprecision, we can derive the use-definition chain for the branch target register, and the target addresses are given by the right hand side of the definitions.

7. Conclusion and Future Work

In this paper, we extend the original strided-interval domain to more precisely track the set of structured numbers, and also define the operations on this extended strided-interval domain. We present the syntax and concrete semantics of the iSAL, which can be used to encode the instructions of different ISAs. In order to achieve generic VSA, we define the abstract semantics for the intermediate language, and discuss how to use it in VSA. We also show an example on using the approach to reconstruct the CFG in the presence of indirect branches.

In the future, we want to try the approach on more architectures, including ARM and PowerPC, and also want to extend iSAL with more static analysis methods.

Acknowledgments

This work is supported in part by the National Science Foundation under Award CNS-1035655.

References

[1] Gogul Balakrishnan, Radu Gruian, Thomas Reps, and Tim Teitelbaum. Codesurfer/x86 – a platform for analyzing x86 executables. In *Proceedings of CC '05*, pages 250–254, 2005.

[2] Gogul Balakrishnan and Thomas Reps. Wysinwyx: What you see is not what you execute. *ACM Trans. Program. Lang. Syst.*, 32(6):23:1–23:84, August 2010.

[3] Sébastien Bardin, Philippe Herrmann, Jérôme Leroux, Olivier Ly, Renaud Tabary, and Aymeric Vincent. The bincoa framework for binary code analysis. In *Proceedings of CAV'11*, pages 165–170, 2011.

[4] Jörg Brauer, René Rydhof Hansen, Stefan Kowalewski, Kim G. Larsen, and Mads Chr. Olesen. Adaptable Value-Set Analysis for Low-Level Code. In *6th International Workshop on Systems Software Verification*, volume 24 of *OpenAccess Series in Informatics (OASIcs)*, pages 32–43, 2012.

[5] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. Bap: A binary analysis platform. In *Proceedings of CAV'11*, pages 463–469, 2011.

[6] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of POPL '77*, pages 238–252, 1977.

[7] Thomas Dullien and Sebastian Porst. Reil: A platform-independent intermediate representation of disassembled code for static code analysis. *Proceeding of CanSecWest*, 2009.

[8] Julian Kranz, Alexander Sepp, and Axel Simon. Gdsl: A universal toolkit for giving semantics to machine language. In Chung-chieh Shan, editor, *Programming Languages and Systems*, volume 8301 of *Lecture Notes in Computer Science*, pages 209–216. Springer International Publishing, 2013.

[9] Thomas Reps, Gogul Balakrishnan, and Junghee Lim. Intermediate-representation recovery from low-level code. In *Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '06, pages 100–111, 2006.

[10] Rathijit Sen and Y. N. Srikant. Executable analysis using abstract interpretation with circular linear progressions. In *Proceedings of MEMOCODE '07*, pages 39–48, 2007.

[11] Alexander Sepp, Bogdan Mihaila, and Axel Simon. Precise static analysis of binaries by extracting relational information. In *Proceedings of WCRE '11*, pages 357–366, 2011.

[12] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. Bitblaze: A new approach to computer security via binary analysis. In *Proceedings of ICISS '08*, pages 1–25, 2008.

[13] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of PLDI '11*, pages 283–294, 2011.