Range and Set Abstraction using SAT

Edd Barrett[1,2] Andy King[3]

School of Computing
University of Kent
Canterbury
England

Abstract

Symbolic decision trees are not the only way to correlate the relationship between flags and numeric variables. Boolean formulae can also represent such relationships where the integer variables are modelled with bit-vectors of propositional variables. Boolean formulae can be composed to express the semantics of a block and program state, but they are hardly tractable, hence the need to compute their abstractions. This paper shows how incremental SAT can be applied to derive range and set abstractions for bit-vectors that are constrained by Boolean formulae.

Keywords: Abstract domains, Numeric domains, Ranges, Satisfiability solving.

1 Introduction

Although the fundamental ideas in abstract interpretation were laid down over thirty years ago [5], abstract interpretation has only entered its industrialisation phase comparatively recently [6]. This new phase is not only characterised by an increased focus on tooling and systems building, but also by work on designing and implementing new abstract domains. For example, domains for improved scalability i.e. the class of weakly-relational domains [15,19], and domains that better match the structure of real programs i.e. symbolic decision trees that correlate the relationship between status flags and numeric

[1] The authors would like to thank Thomas Schilling for his comments on SAT solving, Ralph Corderoy for support on CNF conversion, Jörg Brauer for discussions on minimising bit-vectors, Martin Ellis for code reviews and Stephen Fendyke for sharing his knowledge of interpolation software. We also thank the anonymous reviewers for their comments.

[2] Email: e.barrett@kent.ac.uk

[3] Email: a.m.king@kent.ac.uk

variables [1]. This paper focuses on relating status flags to numeric abstractions that are ranges and sets. Blanchet *et al* [1] illustrates the need for mixed symbolic and numeric abstractions with pseudo-code that is given below:

```
B := (X = 0);
if (!B) Y := 1/X;
```

This code is correct in that sense that it does not give a division by zero error if `X = 0`, but to deduce this it is necessary to track the relationship between `B` and `X`. The authors state:

> "In order to deal precisely with those examples, we implemented a simple relational domain consisting in a decision tree with leaf an arithmetic abstract domain. The decision trees are reduced by ordering Boolean variables and by performing some opportunistic sharing of sub-trees. The only problem with this approach is that the size of a decision tree can be exponential in the number of Boolean variables, and the code contains thousands of global ones"

The problem of relating Booleans to numeric values is particularly acute in binary reverse engineering, though in this context the Booleans are CPU status flags. Binary reverse engineering is the problem of figuring out what a program does from its executable. This is a necessary step when performing, for example, a security audit on code where licensing restrictions prevent access to the source. Motivated by such problems in security, there has been recent interest in recovering the control flow graph (CFG) from a binary [12]. The problem here is the so-called chicken-and-egg problem [8]: to derive the CFG it is necessary to trace values and indirect addresses that occur in registers. However, in order to trace the values in registers, the CFG is required. Kinder resolves this cyclic dependency by applying a constant propagation analysis in conjunction with a CFG that itself grows monotonically as the analysis proceeds [12]. He illustrates these ideas with an idealised assembler language. In practice the problem is considerably harder to solve, partly because of the problem of relating status flags to ranges. To illustrate, consider the following x86 assembler code for a switch table:

```
mov eax, [ebp-0x8]   ; eax := *(ebp - 8)
sub eax, 0x2         ; eax := eax - 2
cmp eax, 0x5         ; CF := (0 =< eax < 5)
                     ; ZF := (eax = 5)
ja  0xd8             ; JMP if CF = 0 and ZF = 0
jmp [0x8048a0c + eax*4]
```

To determine the CFG it is necessary to ascertain that `eax` $\in [0, 5]$ when the indirect jump is reached. This range information, and the table itself, permits

the CFG to be over-approximated. However, inferring the range on `eax` itself requires careful reasoning about the value of the carry (`CF`) and zero (`ZF`) flags — a problem which is analogous to that addressed by Blanchet *et al* [1].

Very recently it has been shown how Boolean formulae can be applied to derive transfer functions for range analysis of AVR micro-controller code [2]. In this work, the semantics of a block of code are represented as a Boolean formula, which is then abstracted with octagons [15] and affine equations [11] so as to derive a transfer function that is a system of guarded updates. In this paper we show how Boolean formulae can be applied directly in the analysis itself in an analogous way to decision trees. However, unlike the decision tree approach, we do not enforce a canonical representation and thereby finesse the size problems that are associated with such data-structures [3]. Instead, we express the semantics of a block as a single propositional formula which can readily be derived with bit-blasting techniques [13]. This formula encodes all the relationships between all the registers and all the status flags, albeit at bit-level granularity. Abstraction is then applied to the formula to extract range invariants that always hold whenever the block is encountered, ultimately allowing the control flow to be recovered. Furthermore, we may impose range information on entry to the block and observe ranges at the exit, again by applying abstraction. This is similar in spirit to work on best transformers [18], but from a propositional stance. In this paper, we focus on how to abstract Boolean formula for range information. Specifically, we make the following contributions:

- We show how to efficiently extract range information for a bit-vector constrained by a Boolean formula where the vector is interpreted as an integer. To be precise, we show how to compute the smallest range that includes all the values the integer can assume, and hence the best over-approximation.
- We show how to refine the range abstraction technique in order to discover boundaries within the range that partition it into a set of ranges. This technique relies on computing over- and under-approximations of the vector in an alternating fashion. This process ultimately converges onto a set S that exactly describes the values of the vector. However, if desired, this process can be terminated prematurely, after n steps, to compute a set that approximates the values of the vector. To be precise, if n is odd then an over-approximation is computed (a superset of S) otherwise if n is even then an under-approximation is found (a subset of S).
- We show how these techniques dovetail with incremental SAT and provide experimental results which suggest that the techniques are viable.

2 Range Abstraction

In order to infer the range of values that a bit-vector $\boldsymbol{x}$ can assume when constrained by a given Boolean formula f – that is, compute a range abstraction for $\boldsymbol{x}$ and f – the maximum and minimum values of $\boldsymbol{x}$ need to be determined. This can be achieved by applying a SAT solver in conjunction with *blocking clauses*. For instance, suppose a SAT solver is applied to find a solution of f under which the propositional variables $\boldsymbol{x} = \langle x_0, \ldots, x_{n-1} \rangle$ are bound to the truth values $b_0, \ldots, b_{n-1} \in \{0, 1\}$. A blocking clause $c = \vee_{i=0}^{n-1} y_i$ is defined by putting $y_i = x_i$ if $b_i = 0$ and $y_i = \neg x_i$ otherwise. Thus any solution to c differs from the truth values $b_0, \ldots, b_{n-1}$ on at least one b_i. The force of this is that the formula $f \wedge c$ excludes the previously found solution. By repeating this technique it is possible to enumerate all solutions, hence all values that $\boldsymbol{x}$ can assume, from which the maximum and minimum can be extracted. The limitation of this technique is that the number of invocations of the solver is linear in the number of solutions (which may be large) and moreover, the size of the SAT instance grows as blocking clauses are added.

2.1 Computing the Minimum

An alternative approach is given in Figure 1 which presents an algorithm for computing a minimum model that requires only n calls to a SAT solver. If the Boolean flag $s = 1$ then the bit vector $\boldsymbol{x} = \langle x_0, \ldots, x_{n-1} \rangle$ is interpreted as a signed integer, represented using two's complement, where x_{n-1} is the sign bit and x_0 is the least significant bit. If $s = 0$ then the bit vector $\boldsymbol{x}$ is interpreted as an unsigned integer. The function *minimum* returns the minimum value expressed as binary vector $\boldsymbol{k} \in \{0, 1\}^n$.

Consider first the unsigned case that is handled in the else branch of the loop body. The bits of $\boldsymbol{k}$ are computed in reverse order: the high bit first and the low bit last. On each iteration of the loop, f is tested to see whether it possesses a solution in which the bit $\boldsymbol{x}[n - |\boldsymbol{k}| - 1]$ is assigned to 0. If so, then the minimum value of $\boldsymbol{x}$ has a 0 in this bit position, hence 0 is prepended to $\boldsymbol{k}$. If not, then every solution of $\boldsymbol{x}$ (including the minimum) has a 1 in this position, hence 1 is prepended to $\boldsymbol{k}$. Note that as the loop progresses, f is itself modified so as to clamp the high bits of $\boldsymbol{x}$ to the high bits of the partially computed minimum $\boldsymbol{k}$.

The signed case proceeds analogously except for the very first iteration which computes the sign bit of the minimum. If f has a solution with $\boldsymbol{x}[n-1]$ assigned to 1, then the minimum is negative, which is reflected by setting $\boldsymbol{k}$ to the unary vector $\langle 1 \rangle$, so as to record the sign of the minimum. Otherwise, the minimum is non-negative, hence $\boldsymbol{k}$ is set to $\langle 0 \rangle$. Setting s to 0 ensures that all subsequent loop iterations deduce the lower bits of $\boldsymbol{k}$ in the same manner as in the unsigned case.

```
(1)  func minimum(f, x, s)
(2)      k ← ⟨⟩, n ← |x|
(3)      while (|k| < n)
(4)          if (s)
(5)              if (sat(f ∧ x[n − 1]))
(6)                  f ← f ∧ x[n − 1]
(7)                  k ← ⟨1⟩
(8)              else
(9)                  f ← f ∧ ¬x[n − 1]
(10)                 k ← ⟨0⟩
(11)             endif
(12)             s ← 0
(13)         else
(14)             if (sat(f ∧ ¬x[n − |k| − 1]))
(15)                 f ← f ∧ ¬x[n − |k| − 1]
(16)                 k ← ⟨0⟩ :: k
(17)             else
(18)                 f ← f ∧ x[n − |k| − 1]
(19)                 k ← ⟨1⟩ :: k
(20)             endif
(21)         endif
(22)     endwhile
(23)     return k
(24) endfunc
```

Fig. 1. Computing the minimum value of the bit-vector $\boldsymbol{x}$

2.2 Computing the Maximum

Computing a maximum model is analogous to computing minimum model. The algorithm for computing a maximum can be obtained from the algorithm shown in Figure 1 by:

- Inverting the polarities of $\boldsymbol{x}[n-1]$ on lines 5, 6 and 9;
- Inverting the polarities of $\boldsymbol{x}[n-|\boldsymbol{k}|-1]$ on lines 14, 15 and 18;
- Inverting the truth values prepended onto $\boldsymbol{k}$ on lines 7, 10, 16 and 19.

3 Set Abstraction

Switch tables can in general be hierarchical structures in which a series of tests direct the control into smaller tables that handle indices that are close to one another. Range abstraction alone cannot accurately model such sets of indices and addresses and therefore it is necessary to instead employ set

```
(1)   func set(f, x, s)
(2)       return set(f, x, s, −1)
(3)   endfunc
(4)
(5)   func set(f, x, s, c)
(6)       S ← ∅
(7)       p ← 1
(8)       l ← ⟨0, . . . , 0, s⟩
(9)       u ← ⟨1, . . . , 1, ¬s⟩
(10)      while (value(l, s) < value(u, s) ∧ c ≠ 0)
(11)          l ← minimum(f ∧ (l ≤_s x), x, s)
(12)          u ← maximum(f ∧ (x ≤_s u), x, s)
(13)          if (p)
(14)              S ← S ∪ [value(l, s), value(u, s)]
(15)          else
(16)              S ← S \ [value(l, s), value(u, s)]
(17)          endif
(18)          p ← ¬p
(19)          f ← ¬f
(20)          c ← c − 1
(21)      endwhile
(22)      return S
(23)  endfunc
```

Fig. 2. Computing a set abstraction for the bit-vector $\boldsymbol{x}$

abstraction. Since an n-ary bit-vector $\boldsymbol{x}$ can assume up to 2^n distinct values, the set itself can be large, at least in the pathological case. Therefore, for cautionary reasons, we seek to compute an over-approximation (superset) that keeps the size of the set manageable. As a by-product of this construction, we are also able to compute under-approximations (subsets) of the set of values that bit-vector can assume when constrained by a given Boolean function f.

Figure 2 presents the function *set* for computing a set abstraction for $\boldsymbol{x}$. The Boolean argument s indicates whether $\boldsymbol{x}$ has a signed interpretation. The integer argument c bounds the number of iterations of the loop. Moreover, if c is non-negative and odd then an over-approximation is found whereas if c is non-negative and even then an under-approximation is derived. If c is negative then the algorithm will run to completion and exactly characterise the values of $\boldsymbol{x}$.

The set S, which starts empty, is refined on each iteration of the loop. The vectors $\boldsymbol{l}$ and $\boldsymbol{u}$ are used to further constrain f; these bounds increase and decrease respectively, until either reaching the c threshold triggers premature termination or the bounds $\boldsymbol{l}$ and $\boldsymbol{u}$ cross and an exact description of the set is

found. The special treatment of the most significant bit of $\boldsymbol{l}$ and $\boldsymbol{u}$ on lines 8 and 9 stem from the two's complement representation for the case that $s = 1$. The function *value* is used to interpret a bit vector as a numeric value:

$$\text{value}(\boldsymbol{b}, s) = (1 - 2s)2^{n-1}\boldsymbol{b}[n-1] + \sum_{i=0}^{n-2} 2^i \boldsymbol{b}[i]$$

Each iteration of the loop determines a new minimum ($\boldsymbol{l}$) and maximum ($\boldsymbol{u}$) solution to a SAT instance that is obtained by augmenting either f or $\neg f$ with a formula that imposes a less-than-or-equals relation on $\boldsymbol{x}$. This additional formula prevents the previously found ranges from being rediscovered. Although incremental SAT can be used within the functions *minimum* and *maximum*, the different less-than-or-equal-to relations impede incremental SAT being applied in the function *set*.

For the unsigned case, the relation is formulated propositionally as follows:

$$\begin{aligned}
\langle\rangle \leq_0 \langle\rangle &= \text{true} \\
\langle \boldsymbol{x}[0] \ldots \boldsymbol{x}[n-1] \rangle \leq_0 \langle \boldsymbol{y}[0] \ldots \boldsymbol{y}[n-1] \rangle &= \\
&(\neg \boldsymbol{x}[n-1] \wedge \boldsymbol{y}[n-1]) \vee ((\boldsymbol{x}[n-1] \Leftrightarrow \boldsymbol{y}[n-1]) \wedge \\
&(\langle \boldsymbol{x}[0] \ldots \boldsymbol{x}[n-2] \rangle \leq_0 \langle \boldsymbol{y}[0] \ldots \boldsymbol{y}[n-2] \rangle))
\end{aligned}$$

whereas the signed case is defined thus:

$$\begin{aligned}
\langle\rangle \leq_1 \langle\rangle &= \text{true} \\
\langle \boldsymbol{x}[0] \ldots \boldsymbol{x}[n-1] \rangle \leq_1 \langle \boldsymbol{y}[0] \ldots \boldsymbol{y}[n-1] \rangle &= \\
&(\boldsymbol{x}[n-1] \wedge \neg \boldsymbol{y}[n-1]) \vee ((\boldsymbol{x}[n-1] \Leftrightarrow \boldsymbol{y}[n-1]) \wedge \\
&(\langle \boldsymbol{x}[0] \ldots \boldsymbol{x}[n-2] \rangle \leq_0 \langle \boldsymbol{y}[0] \ldots \boldsymbol{y}[n-2] \rangle))
\end{aligned}$$

On line 11, $\boldsymbol{l}$ is a vector of truth values, hence the formula ($\boldsymbol{l} \leq_s \boldsymbol{x}$) can be partially evaluated to simplify the comparison (and likewise on line 12 for ($\boldsymbol{x} \leq_s \boldsymbol{u}$)). For example consider two 4-bit vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ and suppose $\boldsymbol{y} = \langle 1, 0, 1, 1 \rangle$. Then $\boldsymbol{x} \leq_0 \boldsymbol{y}$ can be reduced to the formula $\neg \boldsymbol{x}[3] \vee (\boldsymbol{x}[3] \wedge \neg \boldsymbol{x}[2] \vee (\boldsymbol{x}[2] \wedge \neg \boldsymbol{x}[1]))$.

3.1 Example

Suppose a 4-bit vector $\boldsymbol{x}$ is constrained by a formula f so that it can only draw an unsigned value from the set $\{1, 2, 3, 5, 6, 8, 9, 12, 13, 15\}$. The table shows how S converges onto this set by alternating between an over-approximation and an under-approximation.

c	p	$\boldsymbol{l}$	$\boldsymbol{u}$	value $(\boldsymbol{l}, 0)$	value $(\boldsymbol{u}, 0)$	S
-1	1	$\langle 0,0,0,0\rangle$	$\langle 1,1,1,1\rangle$	0	15	$\emptyset$
-2	1	$\langle 1,0,0,0\rangle$	$\langle 1,1,1,1\rangle$	1	15	$\{1 \ldots 15\}$
-3	0	$\langle 0,0,1,0\rangle$	$\langle 0,1,1,1\rangle$	4	14	$\{1,2,3,15\}$
-4	1	$\langle 1,0,1,0\rangle$	$\langle 1,0,1,1\rangle$	5	13	$\{1,2,3,5,6,7,8,9,10,11,12,13,15\}$
-5	0	$\langle 1,1,1,0\rangle$	$\langle 1,1,0,1\rangle$	7	11	$\{1,2,3,5,6,12,13,15\}$
-6	1	$\langle 0,0,0,1\rangle$	$\langle 1,0,0,1\rangle$	8	9	$\{1,2,3,5,6,8,9,12,13,15\}$
-7	0	$\langle 0,1,0,1\rangle$	$\langle 1,0,0,1\rangle$	10	7	$\{1,2,3,5,6,8,9,12,13,15\}$ ✓

4 Experimental Results

The minimum/maximum algorithms at the heart of range abstraction and set abstraction amount to solving a series of related SAT problems. This suggests the application of incremental SAT. Incremental SAT is the problem of solving a series SAT instances $\{\wedge F_1, \ldots, \wedge F_k\}$ defined over a common set of variables. Each F_i is a set of clauses, and the consecutive instances are related according to $F_{i+1} = (F_i \backslash G_i) \cup H_i$ where G_i and H_i are sets of clauses that are respectively rescinded and added [10,20]. Incremental SAT is most useful when $|G_i| \ll |F_i|$ and $|H_i| \ll |F_i|$ since then solving $\wedge F_{i+1}$ can take advantage of the clauses learnt when solving $\wedge F_i$, and possibly earlier instances.

In the algorithm given in Figure 1, unit clauses of $\boldsymbol{x}[n-1]$, $\neg\boldsymbol{x}[n-1]$, $\neg\boldsymbol{x}[n-|\boldsymbol{k}|-1]$ and $\boldsymbol{x}[n-|\boldsymbol{k}|-1]$ are added to f at lines 5, 9, 14 and 18 respectively. Conversely, the unit clauses $\boldsymbol{x}[n-1]$ and $\neg\boldsymbol{x}[n-|\boldsymbol{k}|-1]$ are rescinded when the satisfiability questions posed at lines 5 and 14 are found to have a negative answer. (Note that these removal operations are not reflected in the algorithm but are applied in the else blocks that commence at lines 9 and 18.) Thus whenever a new SAT instance is encountered $|G_i| \leq 1$ and $|H_i| = 1$, which suggests that the algorithm is ideal for incremental SAT. Moreover, only unit clauses are added and removed, and this specialised form of incremental SAT is supported with the so-called unit assumptions of the popular MiniSat solver [9]. (Actually, unit assumptions are automatically withdrawn by MiniSat after checking satisfiability and thus those unit assumptions which need to be preserved have to be readded as immutable clauses.)

Of course, incremental SAT is only of value if it actually improves performance. To investigate this, a series of representative Boolean formulae were generated to model multi-level switch tables so as to constrain a bit-vector $\boldsymbol{x}$ of 64 bits to store up to 98 different branch addresses. To investigate scalability, the set abstraction algorithm was applied to switch tables of increasing size where the branch addresses were non-consecutive (this is because typically addresses are 32 or 64 bits in length, meaning that the first byte of an indirect jump address occurs every 4 or 8 bytes). The abstraction algorithm was not terminated prematurely, so as to exercise it fully. The graph shown in

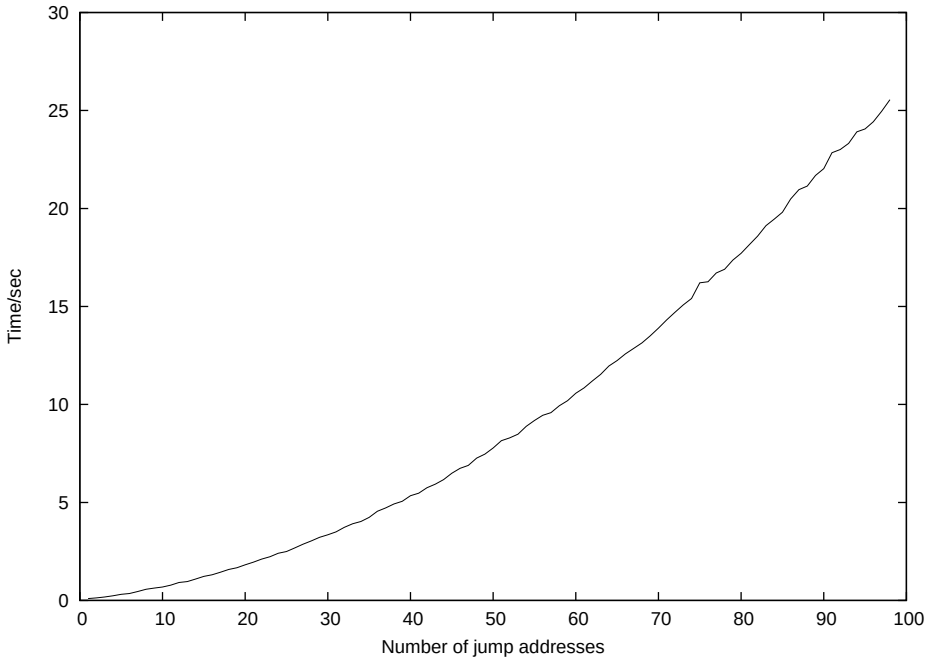

Fig. 3. Satisfiable jump addresses vs. time

Figure 3 suggests that the time to compute the precise set abstraction grows smoothly with the size of the switch table. These timings were generated on a 3GHz x86 machine with 4GB of RAM running Linux.

Interestingly, replacing incremental SAT with a series of independent calls to the same solver gave a slowdown of two orders of magnitude. Thus incremental SAT compensates somewhat for the need to invoke the solver 64 times to compute the minimum/maximum of a 64-bit integer. Figure 4 illustrates the variability in the time required to compute the minima and maxima at different iterations of the set abstraction algorithm. Importantly, the time to compute the minima and maxima do not increase in the latter iterations of algorithm, which one might expect as the solution range diminishes.

It should be emphasised that we report initial results and the efficiency of technique can doubtless be improved by standard tactics such as more refined CNF conversion [17]. Furthermore, by changing the search strategy used in the SAT solver, it may be possible to directly derive the maximum (or minimum) value of a bit vector without deploying n separate (albeit incremental) calls to the solver. Although this would require fundamental changes to the SAT solver itself, the speed-up could be very considerable.

5 Related Work

The question of how to abstract Boolean formulae also arises in the context of deriving transfer functions, specifically those for range analysis [2]. The auto-

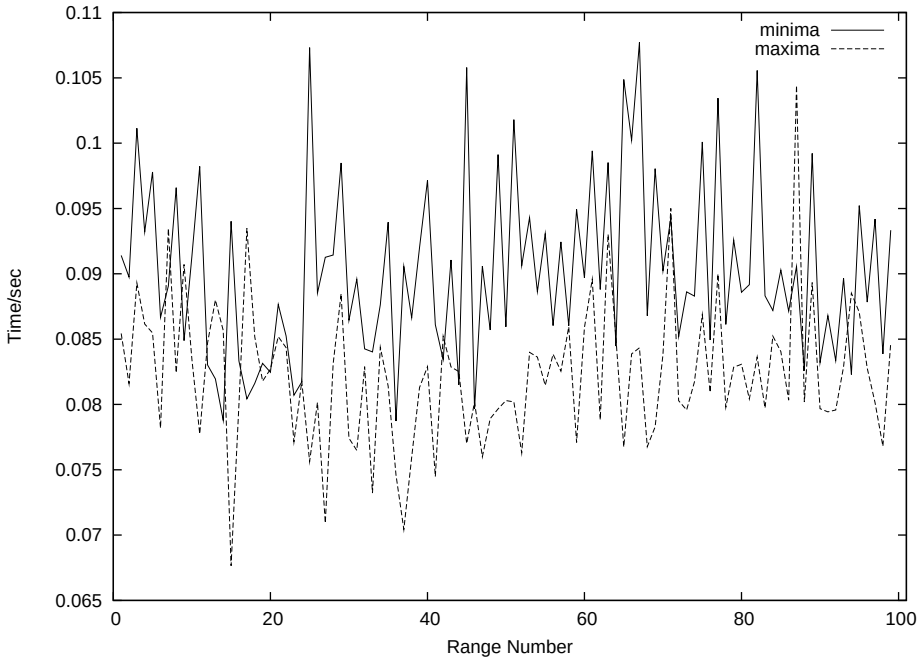

Fig. 4. Time taken to solve minima and maxima of each range.

matic derivation of a transfer function is attractive since it supports reasoning about blocks as a whole thereby improving precision. In range analysis, a transfer function transforms its input ranges to output ranges, and Boolean formulae offer a convenient way of calculating such extreme values since $\forall$-elimination is trivial over this computational domain [13]. There is no reason why the abstraction techniques proposed in this paper can not be deployed to derive octagonal abstractions [15], where one would maximise the expression $x - y$, to derive a constraint of the form $x - y \leq c$.

Cifuentes and Van Emmerik [4] have shown how to compute numeric ranges for switch tables using reverse slicing, however we believe that a more robust approach is to apply bit-blasting with abstraction, as shown in this paper.

Decision procedures have also been applied when computing best transformers [18]. This work is more akin to our own since here the decision procedures are applied within the transfer functions themselves rather than in just deriving them [2]. Moreover, the best transformer work does not address bit-vector encodings, and hence does not consider the associated problem of computing range and set abstractions of their values.

Finally, symbolic decision trees offer an alternative way of relating status flags to numeric variables [1]. This method confers the advantage that the numeric variables can be perfect numbers rather than finite ones which are required for bit-blasting. Symbolic decision trees have traditionally suffered problems of scalability but recent work suggests that this problem can be tackled with judicious widening [7].

6 Conclusion and Future Work

The paper has shown bit-blasting can be combined with range and set abstraction to extract the range of the values that can be used to index a switch table. This is an important step in CFG recovery which itself is important for underpinning other analyses. That such information can be derived automatically from a block is encouraging, particularly as one cannot ensure that the range checks and multi-way branches take a regular recognisable structure.

The method advocated in this paper could be extended to several blocks as an unusual form of path-sensitive analysis in which one accumulates a formula that documents the recent history of computation. This should handle indirect jumps, even in extreme cases where a jump address is passed from one block to another.

We will also investigate as to whether SMT solvers [16] may be used to speed-up our range and set abstraction techniques; it is yet to be seen whether SMT techniques completely finesse the need to operate at bit level granularity.

Of course, CFG reconstruction remains a challenging problem and Linn and Debray [14] have shown that industry standard disassemblers such as IDA Pro can be persuaded to misinterpret large portions of a binary program. This suggests that more robust disassembly techniques must be developed with anti-reversing techniques in mind. Obfuscation techniques such as those based on pre-empting POSIX signal events [14] are likely to remain out of the reach of analysis for some time.

References

[1] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgnei, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI*, pages 196–207, 2003.

[2] J. Brauer and A. King. Automatic Abstraction for Intervals using Boolean Formulae. In R. Cousot and M. Martel, editors, *SAS*, LNCS. Springer, 2010.

[3] R. E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys*, 24(3):318, 1992.

[4] C. Cifuentes and M. Van Emmerik. Recovery of Jump Table Case Statements from Binary Code. *Sci. Comput. Program.*, 40(2-3):171–188, 2001.

[5] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*, pages 238–252, 1977.

[6] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Why does Astrée scale up? *Formal Methods in System Design*, 35(3):229–264, 2009.

[7] P. Cousot, R. Cousot, and L. Mauborgne. A Scalable Segmented Decision Tree Abstract Domain. In *Time for Verification: Essays in Memory of Amir Pnueli*, volume 6200 of *LNCS*. Springer, 2010.

[8] B. De Sutter, B. De Bus, K. De Bosschere, P. Keyngnaert, and B. Demoen. On the static analysis of indirect control transfers in binaries. In *PDPTA*, 2000.

[9] N. Een and N. Sörensson. MiniSat, 2010. www.minisat.se.

[10] J. N. Hooker. Solving the incremental satisfiability problem. *Journal of Logic Programming*, 15(1&2):177–186, 1993.

[11] M. Karr. Affine Relationships Among Variables of a Program. *Acta Informatica*, 6:133–151, 1976.

[12] J. Kinder, F. Zuleger, and H. Veith. An Abstract Interpretation-Based Framework for Control Flow Reconstruction from Binaries. In *VMCAI*, volume 5403 of *LNCS*, pages 214–228. Springer, 2009.

[13] D. Kroening and O. Strichman. *Decision Procedures*. Springer, 2008.

[14] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *ACM Conference on Computer and Communications Security*, pages 290–299, 2003.

[15] A. Miné. The Octagon Abstract Domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[16] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL (T). *Journal of the ACM*, 53(6):937–977, 2006.

[17] D. A. Plaisted and S. Greenbaum. A Structure-preserving Clause Form Translation. *Journal of Symbolic Computation*, 2(3):293–304, 1986.

[18] T. Reps, M. Sagiv, and G. Yorsh. Symbolic Implementation of the Best Transformer. In *VMCAI*, volume 2937 of *LNCS*, pages 3–25. Springer, 2004.

[19] A. Simon, A. King, and J. M. Howe. Two Variables per Linear Inequality as an Abstract Domain. In *LOPSTR*, volume 2664 of *LNCS*, pages 71–89. Springer, 2002.

[20] J. Whittemore, J. Kim, and K. Sakallah. SATIRE: A New Incremental Satisfiability Engine. In *Design Automation Conference*, pages 542–545, 2001.