Refinement-based CFG Reconstruction from Unstructured Programs [⋆]

Sébastien Bardin, Philippe Herrmann, and Franck Védrine

CEA, LIST,
Gif-sur-Yvette CEDEX, 91191 France
first.name@cea.fr

Abstract. This paper addresses the issue of recovering a both safe and precise approximation of the Control Flow Graph (CFG) of an unstructured program, typically an executable file. The problem is tackled in an original way, with a refinement-based static analysis working over finite sets of constant values. Requirement propagation allows the analysis to automatically adjust the domain precision only where it is needed, resulting in precise CFG recovery at moderate cost. First experiments, including an industrial case study, show that the method outperforms standard analyses in terms of precision, efficiency or robustness.

1 Introduction

Motivation. Automatic analysis of programs from their executable files has many potential applications in safety and security, for example: automatic analysis of mobile code and malware, security testing or worst case execution time estimation. This paper addresses the problem of (safe) CFG reconstruction, i.e. constructing a both safe and precise approximation of the Control Flow Graph (CFG) of an unstructured program (typically: an executable file). CFG reconstruction is a cornerstone of safe unstructured program analysis: if the recovery is unsafe, subsequent analyses will be unsafe too; if it is too rough, they will be blurred by too many unfeasible branches and instructions.

Challenges. Such an approximation is difficult to obtain because of dynamic jumps, i.e. jump instructions whose target expression is resolved at run-time and may vary from one execution to the other, leading to vicious circles between value analysis and CFG reconstruction. Unfortunately, dynamic jumps are ubiquitous in native code programs: they may be introduced at compile-time either for efficiency (`switch` in C) or by necessity (return statements, function pointers in C, virtual methods in C++, etc.). Two reasons make CFG reconstruction even more challenging. First, real-life unstructured programs (executable files) contain many junk instructions, i.e. unreachable but well-defined instructions. Since junk instructions cannot be easily recognised when encountered, a small loss of precision on jump targets will often result in an unbearable noise propagation in the analysis. Second, there is no reason why all valid targets of a dynamic jump should follow a nice regular pattern. Indeed they are just addresses in the executable code, often arbitrarily assigned by a compiler. Hence any analysis based on

[⋆] Work partially funded by ANR (grants ANR-05-RNTL-02606 and ANR-08-SEGI-006).

popular domains (i.e. convex domains possibly enhanced with congruence information) will introduce many false targets. For example, consider an instruction `cgoto(x)` with $\texttt{x} \in \{1355, 1356, 2126\}$: such an analysis cannot recover better than $\texttt{x} \in [1355..2126]$, reporting 99% of false targets.

Related approaches. Many works have been developed since the 80's for the CFG reconstruction of *structured programs* (cf. Control Flow Analysis [22]), focusing on control mechanisms such as high-order functions, dynamic methods or pointer functions. Moreover, many static analysers targeting programs written in Java or C commonly face the CFG reconstruction issue. However, there are a few significant differences with the case of unstructured programs: the "jump" information is clearly identified and separated from data information (syntactically or through types), it cannot be arbitrarily manipulated (syntactic restrictions or clean encapsulation) and the set of all possible jump targets is statically known and usually small. Thus, in Java or C a safe approximation of the CFG can often be recovered with an abstract propagation based on finite sets of constants (k-sets) limited to a few relevant and easy-to-find variables. Unfortunately none of these restrictions hold on unstructured programs.

Only very few works address the CFG reconstruction of *unstructured programs* (simply referred to as CFG reconstruction in the rest of the paper). Reps *et al.* [5, 7, 8] develop dedicated heuristics based on strided intervals, affine relationships discovery and local variable identification to avoid rough abstraction of jump targets. Their results are implemented in CODESURFER/X86 [1]. The tool JAKSTAB [18] developed by Kinder and Veith is based on k-set propagation [19]. Experiments reported by the authors show that while each approach performs much better than current industrial tools, like IDA PRO [24], both techniques still recover many false targets. Especially, strided intervals cannot capture precisely sets of jump targets, and k-sets are too sensitive to their cardinality bound, potentially leading to either imprecise or expensive analyses.

Our approach. We make the following observations. First, while k-sets are considered as a crude and/or costly domain in most static analysis settings, we think that it is the only abstract domain well-suited for representing sets of dynamic targets, as long as the cardinality bound is large enough. Second, CFG reconstruction seems really well-suited to lazy reasoning: on the one hand, target expressions must be tracked very precisely since a small precision loss there may have a dramatic impact on the whole analysis, on the other hand, we claim that in most realistic settings only a few facts need to be tracked to solve dynamic jumps. The last point may be exploited to reduce the cost of k-set propagation without affecting preciseness.

Contribution. The main results of this paper are twofold. First, we introduce a new framework of Value Analysis with Precision Requirements (VAPR) and show how precise CFG reconstruction fits this framework. We then propose an original refinement-based procedure to solve this problem. The procedure is built on two main steps: a forward k-set propagation with local cardinality bounds (ranging from 0 up to a given parameter $Kmax$), and a refinement step controling these cardinality bounds, driven by backward requirement propagation. The procedure is sound and terminates, moreover it is complete relative to standard k-set propagation on a class of non-trivial programs.

Second, this framework has been implemented in a prototype and first experiments demonstrate a very precise and efficient CFG reconstruction, with great robustness to the initial parameter $Kmax$. The approach performs better than other straightforward analyses, demonstrating that it is really the combination of local refinement and k-sets which leads to precision and efficiency. The prototype has also been successfully tested against an industrial case study, proving the scalability of the approach.

Outline. The rest of the paper is structured as follows. Section 2 presents notations and background. Section 3 describes the Value Analysis with Precision Requirements (VAPR) problem and the Propagate-and-Refine (PaR) procedure to solve it. Section 4 studies the relative completeness of the approach. Section 5 describes a prototype implementation and experiments. Finally, Section 6 discusses related work and Section 7 provides a conclusion and directions for future works.

2 Preliminaries

Unstructured programs. An unstructured program P is a tuple (L, V, A, T, l_0) where: $L \subseteq \mathbb{N}$ is the finite set of code addresses, V is the finite set of program variables, A is the finite set of arrays, $T : L \mapsto I$ is a partial function mapping code addresses to program instructions $i \in I$, and l_0 is the initial code address. Arrays have a statically known size. For the sake of simplicity, all variables and arrays range over the set of natural numbers $\mathbb{N}$, however all results of the paper are easy to adapt to any basic data type. The set of extended variables, i.e. either variables $v \in V$ or array elements $a[k]$ with $a \in A$ and $k \in \mathbb{N}$, is denoted by $V^{\#}$. Program instructions are composed of: assignments v:=e, static jumps `goto` l, branching instructions `ite`($cond$,l_1,l_2), dynamic jumps `cgoto`(v) and halting instruction `stop`, where $l \in L$, $v \in V^{\#}$, $cond$ and e are predicates and expressions built over $\mathbb{N}, V$ and A.

The concrete domain of P is $Dom = \mathbb{N}^{|V^{\#}|}$. The operational semantics of P is given in a standard way by a transition system whose configurations are either pairs $(l, \sigma) \in L \times Dom$ or an error configuration $Undef$. Considering a valuation $\sigma \in Dom$, the value of $v \in V^{\#}$ in σ is denoted by $\sigma(v)$. The successor (l', σ') of (l, σ) through instruction $(l, T(l)) \in L \times I$ is defined straightforwardly for assignments, static jumps and conditional branchings. If $T(l)$ is of the form `cgoto`(v) then $(l', \sigma') = (\sigma(v), \sigma)$. If $T(l)$ is undefined the successor is $Undef$, which has no successor itself. An instruction at address l is well-defined if $T(l)$ is defined. A valid instruction is a well-defined and reachable instruction, a junk instruction is a well-defined but unreachable instruction.

The k-set domains. We follow the basic concepts of abstract interpretation [9]. The abstract domains considered in this paper are based on k-sets, i.e. finite sets of constant values (in $\mathbb{N}$). The lattice (or domain) of all k-sets of cardinality at most k (extended with $\top = \mathbb{N}$, and $\bot$ denotes $\emptyset$) is denoted by $KSET(k)$. An abtract value $\widehat{d} \in KSET(k)$ is either a k-set with at most k values or $\top$. For example, $KSET(1)$ is the lattice of constant propagation and $KSET(0)$ is $\{\bot, \top\}$. By convention $KSET(\infty)$ will denote the set of all k-sets (no cardinality bound) extended with $\top$. A pair $(l, v) \in L \times V^{\#}$ is called a *location*. A memory state is a map M from locations to k-sets.

Running example. Here is a small example illustrating the "chicken-or-egg" issue between value analysis and CFG reconstruction of unstructured programs. Because of the mod operation at line 1, possible values of x at the beginning of line 2 are $\{0,1,2\}$, then possible targets of the dynamic jump at line 3 are $\{10,20,30\}$. Let us also suppose that code address 11 contains a junk instruction goto 2. We consider a CFG recovery performed with intervals and standard widening (denoted $\triangledown$). Starting from $(1,x) = \top$, abstract value $(2,x) = [0..2]$ is propagated, yielding the potential targets $[10..30]$ at line 3. Values are propagated to valid instructions at line 10, 20 and 30. The $Undef$ configuration is propagated for all other potential targets but 11. However, the abstract value at line 3 is also propagated to the junk instruction at line 11 (carrying $x = 11$), and propagated back to line 2. Then $(2,x)$ becomes $[0..2]\triangledown[11] = \top$ and target jump $(3,x)$ becomes $\top$, leading to a very imprecise CFG reconstruction.

```
1 : assign x := x mod 3 ; goto 2
2 : assign x := 10(x+1) ; goto 3
3 : cgoto(x)
10 : stop
11 : goto 2   /* junk */
20 : stop
30 : stop
```

Fig. 1. Running example Foo

Moreover, the best possible abstract value for the target expression using any convex domain is $x \in [10..30]$, still yielding 18 false targets out of 21 recovered values.

3 The Propagate and Refine procedure

3.1 Value analysis with precision requirements (VAPR)

Given a program P, an atomic precision requirement is a pair $(l,v) \in L \times V^{\#}$, denoted by $\varphi\langle l,v\rangle$. A memory state M satisfies $\varphi\langle l,v\rangle$ if $M(l,v) \neq \top$, denoted by $M \models \varphi\langle l,v\rangle$. The definition is extended to any set $\mathcal{C}$ of precision requirements: $M \models \mathcal{C}$ iff M satisfies every $\varphi\langle l,v\rangle \in \mathcal{C}$. A location (l,v) is faulty for $(M,\mathcal{C})$ whenever $M(l,v) = \top$ and $\varphi\langle l,v\rangle \in \mathcal{C}$. Given a program P and a set of precision requirements $\mathcal{C}$, the problem of *Value Analysis with Precision Requirements* (VAPR) on $(P,\mathcal{C})$ consists in computing an over-approximation M of the collecting semantics of P such that $M \models \mathcal{C}$.

Precise CFG reconstruction can be achieved through VAPR as follows: an unstructured program P is transformed into a VAPR problem by adding a requirement $\varphi\langle l,v\rangle$ for each instruction $(l, cgoto\ v)$ in P, ensuring at least that no jump target will evaluate to $\top$. It turns out that this simple kind of requirements combined with k-sets leads to precise results in practice (cf. Section 5).

3.2 Basic intuitions on the Propagate and Refine procedure

A refinement-based procedure for VAPR is sketched here, a detailed description is given in Section 3.3. The procedure returns either an invariant of the program or a FAIL message. It follows a “propagate-and-refine” scheme: a domain precision is attached to each location ; abstract values are forward-propagated almost as usual; when a precision requirement is violated, a backward refinement mechanism takes place from faulty locations to remove the violation by increasing the precision of some (hopefully relevant) domains; if at least one domain is improved, the refinement succeeds and propagation is restarted with the new domains and the initial memory state, otherwise the procedure returns FAIL. The procedure is sound, in the sense that in case of success the invariant respects the precision requirements, and terminates (cf. Section 3.3). The main ingredients of the procedure are the following:

1- Each location (l, v) is associated with a $KSET$ domain, whose size (denoted by $D(l, v)$) is bounded by a global parameter $Kmax$. Implicit casts are performed during propagation between locations with different domains: computation is first done with maximal precision, then adjusted to the cardinality bound of the destination (the cast function returns identity if the k-set size matches the new bound, $\top$ otherwise). Casts allow to loosen precision on some (hopefully irrelevant) locations and gain efficiency.

2- During propagation, $\top$ values are tagged with labels (namely, $\top$-labels) recording their origins. Especially, the labelled value $\top_{init}$ denotes initial $\top$ values, while $\top_{\langle d_1,\ldots,d_q\rangle}$ indicates a $\top$ value coming from the abstraction of a k-set $\{d_1, \ldots, d_q\}$ to $\top$ (called a $\top$-abstraction). $\top$-abstractions are very important in the procedure: these sources of precision loss could have been avoided with a domain bound large enough if $Kmax \geq q$. Values $\top_{\langle d_1,\ldots,d_q\rangle}$ allow the analysis to pinpoint such $\top$-abstraction and indicate the minimal domain bound required. Finally, $\top_*$ denotes $\top$ values coming from propagation of $\top_{init}$ or $\top_{\langle d_1,\ldots,d_q\rangle}$.

3- Refinement starts from faulty locations and follows backward data-dependencies to find all ancestors (l, v) evaluating to some $\top_{\langle d_1,\ldots,d_q\rangle}$ with $q \leq Kmax$. In that case, $D(l, v)$ is increased to q (we say that it is *corrected*). The propagation chain stops on $\top_{init}$ and $\top_{\langle d_1,\ldots,d_q\rangle}$ with $q > Kmax$ since they cannot be corrected. Refinement fails when no domain has been increased.

4- A journal logs which target jumps, conditional branches and array indexes have been used for each instruction during value propagation. This information allows to prune requirement propagation, playing a major role in refinement accuracy.

Refinement. The refinement step is the crucial point of the procedure: if too many location domains are refined the procedure will be inefficient, if not enough location domains are refined the procedure will often fail while standard propagation over $KSET(Kmax)$ would have succeeded. In a sense, we would like to discover along the propagation trace where the relevant loss of precision occurs first, correct the domain precision accordingly, and restart the propagation from that point. However, this temporal reasoning is not practical since it would imply to log huge traces of (large) memory states. Instead, the standard propagation mechanism is enriched with a little additional information ($\top$-labels, journal) so that the costly temporal reasoning on traces is replaced by a lighter spatial reasoning on the current memory state (cf. Figure 2). This reduction is not perfect and some information is lost in the general case, however it appears to be very precise and efficient in practice (cf. Section 5), and the reduction is perfect for certain classes of unstructured programs (cf. Section 4).

Figure 2 exemplifies the refinement mechanism in a simple case. Each code address L_1 to L_5 is annotated with the domain bound and abstract value of the relevant variable for the next instruction. Domain bounds are written Dx, Da and Db. At step 1 of the propagation, value $\top_{\langle 1,2\rangle}$ is propagated to (L_3, x) by incoming transitions from L_1 and L_2 because Dx < 2. This $\top$-abstraction leads to (L_5, b) evaluating to $\top$ at step 3. Let us assume that this is a requirement violation. Then a temporal reasoning would amount to go back along the execution trace to find the source of precision loss, at location (L_3, x) in step 1. Thanks to additional information, a simple backward propagation on data-dependencies on the local memory state (step 3) allows to detect that the requirement violation comes from (L_3, x), whose domain bound should be at least 2.

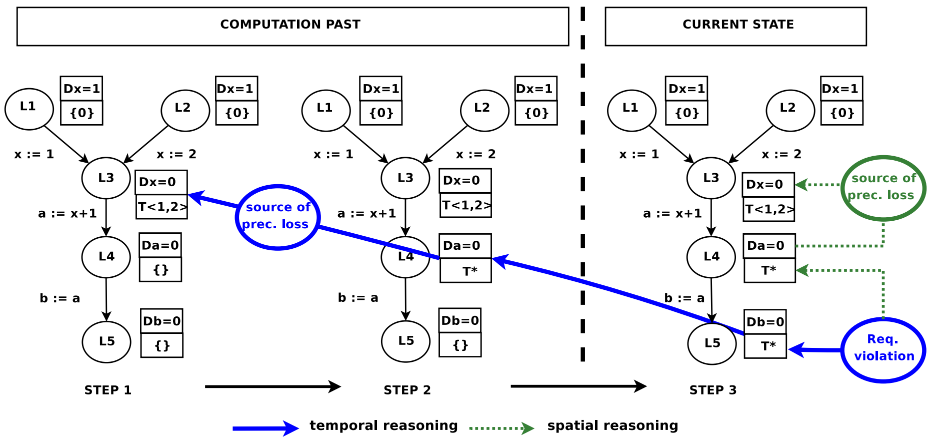

Fig. 2. Temporal reasoning (along the execution trace) vs spatial (local) reasoning

Running example. The program `Foo` of Figure 1 is turned into a VAPR problem by adding a single precision requirement $\mathcal{C} = \{\varphi\langle 3, x\rangle\}$. Let M be the memory state built during the analysis, and D the function which maps each location to its domain cardinality bound. At the beginning of the procedure, $M(1, x) = \top_{init}$ and all other locations evaluate to $\bot$. Moreover, all domain bounds are set to 0. A first propagation step takes place, ending with the following memory state : $M = \{(1, x) \mapsto \top_{init}, (2, x) \mapsto \top_{\{0,1,2\}}, (3, x) \mapsto \top_*, \}$ ($\bot$ values are omitted), and a requirement violation is detected for $(3, x)$. The refinement step takes place from $(3, x)$. Since $(3, x)$ cannot be corrected (it is only a simple $\top_*$), the requirement is propagated to its unique predecessor $(2, x)$. The abstract value is $\top_{\{0,1,2\}}$ and $D(2, x) = 0$. If $Kmax < 3$ then $(2, x)$ cannot be corrected and the procedure fails since no location has been corrected. Note that it is indeed impossible to satisfy the requirement with $Kmax$ strictly less than 3. Otherwise ($Kmax \geq 3$) $D(2, x)$ is set to 3 and the propagation is restarted, leading to $M = \{(1, x) \mapsto \top_{init}, (2, x) \mapsto \{0, 1, 2\}, (3, x) \mapsto \top_{\{10,20,30\}}\}$, and a requirement violation is again detected for $(3, x)$. The refinement step goes on and $(3, x)$ is directly corrected, $D(3, x)$ is set to 3 and the propagation is restarted. The memory state obtained is $M = \{(1, x) \mapsto \top_{init}, (2, x) \mapsto \{0, 1, 2\}, (3, x) \mapsto \{10, 20, 30\}, (10, x) \mapsto \top_*, (20, x) \mapsto \top_*, (30, x) \mapsto \top_*\}$. No violation is reported, the procedure succeeds.

3.3 The PaR procedure for VAPR

The Propagate and Refine procedure (PaR) for VAPR is described precisely hereafter. Procedure 1 is the top-level procedure, based on the propagation step (Procedure 2) and the refinement step (Procedure 3), built itself on requirement propagation (Procedure 4). In the following procedures, P is an unstructured program (type $\mathcal{P}$), $\mathcal{C}$ is a set of requirements (type $\mathcal{R}$), $S \subseteq L \times V^{\#}$ is a set of locations (type $\mathcal{S}$), $D : L \times V^{\#} \mapsto \mathbb{N}$ is a domain map (type $\mathcal{D}$), $M : (l, v) \in L \times V^{\#} \mapsto KSET(D(l, v))$ is a memory state (type $\mathcal{M}$), and e is a program expression (type $\mathcal{E}$). Expressions are built on integer values, variables, array-reads and unary / binary operators. The notation $?$*Type* indicates that a procedure returns either a value of type *Type* or a special value $FAIL$.

Propagation (see Procedure 2). The forward (flow-sensitive) value propagation (PROPAGATE*) differs from a standard one on a few points: (1) faulty locations are systematically discarded and recorded (in output S), (2) implicit casts are performed during propagation between locations with different domains, (3) $\top$-labels are propagated as explained hereafter, (4) value propagation also updates the journal.

Journal. The journal is updated in procedure PROPAGATE* (Procedure 2) and read in requirement propagation (Procedure 4) to prune valid but irrelevant data-dependencies. For the sake of clarity, the journal is split into two different maps $H_1 : L \times I \times \mathcal{E} \rightarrow KSET(\infty)$ (type $\mathcal{H}_1$) and $H_2 : L \times I \times \mathbb{B} \rightarrow \mathbb{B}$ (type $\mathcal{H}_2$), where $\mathbb{B}$ is the set of boolean values. An instruction is *fired* when it is used in value propagation. The value $H_1(l, i, e)$ stores the maximal k-set at which expression e has been evaluated when instruction (l, i) was fired. This is used to record relevant jump targets and array indexes. The value $H_2(l, i, b)$ stores whether or not the conditional instruction (l, i) has been fired with condition evaluating to b.

$\top$-labels. The set F of $\top$-labels has four kind of elements: f_{init} for initial $\top$ values, $f_{\langle d_1, d_2, \ldots \rangle}$ for the $\top$-abstraction of $\{d_1, d_2, \ldots\}$, f_* for other $\top$-values, and $f_\perp$ to denote the absence of $\top$-value. Hence each location is associated to a pair in $KSET(\infty) \times F$. F is equipped with the following order: $f_\perp$ is the minimal element, all other elements are incomparable. Propagation rules for $\top$-labels are the following: f_{init} and $f_{\langle \ldots \rangle}$ are kept in place (never overwritten), and they are propagated as f_*. Strictly speaking, F is not a lattice. However, since $\top$-labels are not considered for concretization, k-set propagation or inclusion testing, they can be seen as an additional information not affecting the main k-set propagation. From a theoretical point of view, there are now several minimal fixpoints of the program only diverging on their $\top$-labels. In the following $f_\perp$ is always omitted, and for other $f_\# \in F$, $(\top, f_\#)$ is written $\top_\#$.

Theorem 1. *Procedure PaR(P,$\mathcal{C}$) terminates and is sound, i.e. it returns either FAIL or a safe approximation M of the collecting semantics of P such that $M \models \mathcal{C}$.*

Proof. The proof for soundness is straightforward since all requirements are checked to hold before procedure PaR returns an invariant of the program. For termination, note first that procedures PROPAGATE* and REFINE terminate: for PROPAGATE*, lattices based on cartesian products of k-sets satisfy the Ascending Chain Condition, and $\top$-label propagation terminates because a $\top_{\langle \ldots \rangle}$ cannot be overwritten ; for REFINE, each $(l, v) \in L \times V^{\#}$ is processed at most once in the procedure. Moreover, PaR can go through label `restart` only when a domain refinement occurs, which can happen at most $|L| \times |V^{\#}| \times Kmax$ times, ensuring termination. □

The procedure runs in polynomial-time: there are at most $|L| \times |V^{\#}| \times Kmax$ domain refinements, hence there are at most $|L| \times |V^{\#}| \times Kmax$ steps of propagation-refinement, and both propagation and refinement are polynomial-time procedures.

```
**PaR** : $(P : \mathcal{P}) \times (\mathcal{C} : \mathcal{R}) \mapsto ?\mathcal{M}$
  parameter: a maximal bound for k-sets $Kmax$
  input: a program $P = (L, V, A, T, l_0)$ and a set of precision requirements $\mathcal{C}$
  output: an invariant $M$ of $P$ covering $(l_0, \top^{|V|^{\#}})$ such that $M \models \mathcal{C}$, or FAIL
——
  $D$ := map($(l, v) \mapsto\ 0$ )   /* $D : L \times V \mapsto \mathbb{N}$ maps $(l, v)$ to a size of k-set */
  label restart
  $M$ := map($(l, v) \mapsto\ if\ l = l_0\ then\ \top_{init}\ else\ \bot$ )
    /* $M$ : memory map, $M(l, v)$ is a k-set whose size is bounded by $KSET(D(l, v))$ */
  ($H_1$,$H_2$) := empty-journal  /* $H_1$ always evaluates to $\bot$, $H_2$ always evaluates to $false$ */
  $S$ := $\emptyset$    /* set of faulty locations */
  ($M$, $H_1$, $H_2$, $S$) := PROPAGATE$^*$($P$, $\mathcal{C}$, $M$, $D$)
  if $S \neq \emptyset$ then
    case REFINE($P$, $D$, $M$, $H_1$, $H_2$, $S$) of
      | FAIL : return FAIL
      | RES($D'$) : $D$ := $D'$; goto restart    /* domain is refined */
    end case
  end if
  return M
```

Procedure 1: PaR, global scheme

```
**PROPAGATE**$^*$ : $(P : \mathcal{P}) \times (\mathcal{C} : \mathcal{R}) \times (M : \mathcal{M}) \times (D : \mathcal{D}) \mapsto \mathcal{M} \times \mathcal{H}_1 \times \mathcal{H}_2 \times \mathcal{S}$
  input: a program $P = (L, V, A, T, l_0)$, a domain map $D$, a memory state $M$,
       a set of precision requirements $\mathcal{C}$
  output: updated version of $M$, a journal $(H_1, H_2)$ and the set of faulty locations for $(M, \mathcal{C})$
——-
  $S$ := $\emptyset$ ; $(H_1, H_2)$ := empty-journal
  While $M$ not a fixpoint of $P$ do
      forall $(l, T(l)) \in L \times I$ of $P$ do
        if for all $\varphi\langle l, v\rangle \in \mathcal{C}$, $M(l, v) \models \varphi\langle l, v\rangle$ then
          $M'$ := eval($(l, i), M, D$)     /* $D$ is needed for cast between domains */
          $(H_1, H_2)$ := update-journal($(l, i), M, D, H_1, H_2$)
          $M$ := $M \sqcup M'$
        end if
      end forall
  end while
  $S$ := $\{(l, v) \in L \times V^{\#} | M(l, v) \not\models \mathcal{C}\}$
  return ($M$, $H_1$, $H_2$, $S$)
——-
journal updates : $H_1$ and $H_2$ are updated according to the following events
 – when instruction $(l, i) = (l, cgoto\ v)$ is fired, then $H_1(l, i, v) := M(l, v)$
 – when $(l, i) = (l, ite(p, l_1, l_2))$ is fired for $p$ evaluating to $b$, then $H_2(l, i, b) := true$
 – when $(l, i) = (l, a[e] := ...)$ is fired, then $H_1(l, i, e) := M(l, e)$
 – when $(l, i) = (l, ... := f(a[e], v_1, ..., v_n))$ is fired, then $H_1(l, i, e) := M(l, e)$
```

Procedure 2: Value propagation

```
**REFINE** : $(P : \mathcal{P}) \times (D : \mathcal{D}) \times (M : \mathcal{M}) \times (H_1 : \mathcal{H}_1) \times (H_2 : \mathcal{H}_2) \times (S : \mathcal{S}) \mapsto ?\mathcal{D}$
  parameter: a maximal bound for k-sets $Kmax$
  input: a program $P = (L, V, A, T, l_0)$, a domain map $D$, a memory state $M$,
       a journal $(H_1, H_2)$, a set of locations $S = \{(l_i, v_i)\}$ such that $M(l_i, v_i) = \top$
  output: return updated version of $D$, or FAIL
————
  success := false
  SET := **propagate-req**$^*$($P$, $S$, $M$, $H_1$, $H_2$)
  **For all** $(l, v) \in$ SET such that $M(l, v) = \top_{\langle d_1,\ldots,d_{\mathbf{q}}\rangle}$ **do**    /* clearly $D(l, v) < q$ */
    **if** $q \leq Kmax$ **then** $D(l, v) := q$, success := true **end if**
  **end for**
  **if** success **return** RES($D$) **else return** FAIL **end if**
```

Procedure 3: Refinement

```
**propagate-req**$^*$ : $(P : \mathcal{P}) \times (S : \mathcal{S}) \times (M : \mathcal{M}) \times (H_1 : \mathcal{H}_1) \times (H_2 : \mathcal{H}_2) \mapsto \mathcal{S}$
  input: a program $P$, a memory state $M$, a journal $(H_1, H_2)$, a set of faulty locations $S$
  output: set of locations influencing the value of locations in $S$, with pruning based on $(H_1, H_2)$
—-
  **return** lfp of **prop-req-instr** starting from locations in $S$ and using all $(l, i) \in L \times I$ of $P$
—-
**prop-req-instr** : $((l_0, i) : L \times I) \times ((l, v) : L \times V^{\#}) \times (M : \mathcal{M}) \times (H_1 : \mathcal{H}_1) \times (H_2 : \mathcal{H}_2) \mapsto \mathcal{S}$
  /* backward propagation from faulty location $(l, v)$ for instruction $(l_0, i)$ */
  **if** $M(l, v) = \top_{init}$ or $M(l, v) = \top_{\langle\ldots\rangle}$ **then** return $\emptyset$
  **else if** $M(l, v) \neq \top_*$ **then** return $\emptyset$
  **else case** $l_0, i$ **of**    /* we sketch only the most representative cases */
    | $l_0, cgoto(x)$: **if** $l \in H_1(l_0, i, x)$ return $\{(l_0, v)\}$ **else** return $\emptyset$
    | $l_0, ite(p, l_1, l_2)$ with $l = l_1$ : **if** $H_2(l_0, i, true) = true$ **then** return $\{(l_0, v)\}$ **else** return $\emptyset$
    | $l_0, ite(p, l_1, l_2)$ with $l = l_2$ : **if** $H_2(l_0, i, false) = true$ **then** return $\{(l_0, v)\}$ **else** return $\emptyset$
    | $l_0, a[e'] := e, l$ and $v = a[k]$ :
          **if** $\{k\} = H_1(l_0, i, e')$ **then** return $\{l_0\} \times$**prop-req-expr**($e, H_1, l, i$)
          **else if** $k \notin H_1(l_0, i, e')$ **then** return $(l_0, v)$
          **else** return ($\{l_0\} \times$**prop-req-expr**($e, H_1, l, i$)) $\bigcup$ $\{(l_0, v)\}$
          **end if**
  **end case**
  **end if**
—-
**prop-req-expr** : $(e : \mathcal{E}) \times (H_1 : \mathcal{H}_1) \times (l : L) \times (i : I) \mapsto 2^{V^{\#}}$
  **case** $e$ **of**    /* we sketch only the most representative cases */
    | $k \in \mathbb{N}$ : return $\emptyset$    | $v \in V$ : return $\{v\}$    | $a[k]$, $k \in \mathbb{N}$ : return $\{a[k]\}$
    | $op(e_1, e_2)$ : return **prop-req-expr**($e_1, H_1, l, i$) $\cup$ **prop-req-expr**($e_2, H_1, l, i$)
    | $a[e]$ : **if** $H_1(l, i, e) = \top$ **then** return **prop-req-expr**($e, H_1, l, i$)
          **else** return $\{a[k] | k \in H_1(l, i, e)\}$
          **end if**
  **end case**
```

Procedure 4: Requirement propagation

4 Relative completeness

Let $(P, \mathcal{C})$ be a VAPR problem. We denote by $KSET(k)$-PROP the standard forward value propagation built over $KSET(k)$, and by $lfp_{\langle kset(k) \rangle}(P)$ the best abstraction of the collecting semantics of P using $KSET(k)$. PaR is relatively complete if PaR$(P, \mathcal{C})$ with parameter $Kmax$ returns successfully when $KSET(Kmax)$-PROP$(P) \models \mathcal{C}$. Relative completeness indicates how precise the refinement is.

No relative completeness in the general case. Here is a small example demonstrating that PaR is not relatively complete. Let us consider the `Foo2` program in Figure 3 with precision requirement $\mathcal{C} = \{\varphi\langle 5, t\rangle\}$ and $Kmax = 1$. Procedure $KSET(1)$-PROP returns M with $M(l_5, t) = \{100\}$, thus the result satisfies $\mathcal{C}$. However, PaR fails on that program. The first value propagation computes a memory state such that $\{(1, x) \mapsto \top_{init}; (2, x) \mapsto \top_{\langle 1\rangle}; (5, t) \mapsto \top_{\langle 100\rangle}\}$ or $\{(1, x) \mapsto \top_{init}; (2, x) \mapsto \top_{\langle 1\rangle}; (5, t) \mapsto \top_{\langle 200\rangle}\}$, depending on the order of propagation steps in PROPAGATE*. $\mathcal{C}$ is violated, and a refinement is launched. $D(5, t)$ is increased to 1, and the forward propagation goes on. Note that $D(2, x)$ still equals 0. The result satisfies $M(5, t) = \top_{\langle 100,200\rangle}$, again violating $\mathcal{C}$. However, this time $D(5, t)$ cannot be corrected ($Kmax$ is too low), and the backward refinement propagation stops (only constant assignments to $(5, t)$). The refinement procedure fails, causing PaR to fail.

```
1. x:=1, goto 2
2. if x=1 then goto 3 else goto 4
3. t := 100, goto 5
4. t := 200, goto 5
5. cgoto t
```

Fig. 3. The `Foo2` program.

Relative completeness on SVAPR. We consider relations $R \subseteq 2^{Dom} \times 2^{Dom}$ over sets of valuations, variables of the relation being partitioned into input variables (x_i)'s and output variables (x'_i)'s. A relation $R(x_1, \ldots, x_n, x'_1, \ldots, x'_n)$ is *simple* if for all $i \in [1..n]$, there exists $J_i \subseteq [1..n]$ such that (a) the value of x'_i depends only of $\{x_j | j \in J_i\}$, and (b) $card(x'_i) = +\infty$ iff $\bigvee_{j \in J_i} card(x_j) = +\infty$. A function from 2^{Dom} to 2^{Dom} is simple if its associated relation is simple. The class of simple functions comprises operations such as $+, -, abs, \times k$. The $\times$ operation is not simple, since $\mathbb{N} \times \{0\} = \{0\}$.

The class of SVAPRs comprises all VAPRs $(P, \mathcal{C})$ such that there are no conditional branching (only non-deterministic choice), all functions in expressions are simple, and $\mathcal{C}$ comprises a requirement $\varphi\langle l, v\rangle$ for each instruction $(l, cgoto\ v)$ and each array expression $a[v]$ at code address l. Jumps and array expressions are said to be *guarded*.

Theorem 2 (Relative completeness). *PaR is relatively complete on SVAPR.*

The proof is sketched hereafter. In the following, a value $\top_{\langle d_1, \ldots, d_q\rangle}$ is said to be rectifiable if $q \leq Kmax$, the symbol $\top$ is used to denote any special $\top$ value regardless its $\top$-label and a proper k-set denotes a k-set different from $\top$. The general scheme of the proof is to consider an arbitrary execution ρ of PaR leading to a failure (let us imagine that $\varphi\langle l_n, v_n\rangle \in \mathcal{C}$ is violated), and build from it a sequence of propagations in $KSET(Kmax)$ demonstrating that $lfp_{\langle kset(kmax)\rangle}(P)(l_n, v_n) = \top$. It then follows that any safe analysis of P based on $KSET(Kmax)$ cannot satisfy $\mathcal{C}$.

Let $\pi = t_1 \ldots t_n$ be the part of ρ representing the last call to PROPAGATE*. Let us denote by $M_1, \ldots, M_n$ the sequence of successive memory states obtained from

M_0 (initial state) following π. We suppose that $M_n(l_n, v_n) = \top$ while $\varphi\langle l_n, v_n\rangle \in \mathcal{C}$. By definition of π, there is no requirement violation when t_k is applied to M_k, and no predecessor of (l_n, v_n) (by requirement propagation) can be corrected since refinement fails. Let us denote by (M_i')'s the memory states obtained from $M_0 = M_0'$ by a $KSET(Kmax)$ propagation following π. The proof consists in showing that $M_n'(l_n, v_n) = \top$. We first suppose that there are no array expressions nor dynamic jumps along π. The main steps of the proof are the following.

(L1) We can always find (s, l_s, v_s) such that $M_s(l_s, v_s)$ is a non-rectifiable source of the requirement violation, i.e.: (l_s, v_s) is a predecessor of (l_n, v_n), and $M_s(l_s, v_s)$ either evaluates to $\top_{\langle d_1,\ldots,d_q\rangle}$ with $q > Kmax$ and the direct predecessors of (l_s, v_s) in M_{s-1} being proper k-sets, or evaluates to $\top_{init}$. In the first case, the inequality $q > Kmax$ is deduced from the failure of the refinement.

(L2) Proper k-sets computed in PaR along π are conserved in $KSET(Kmax)$, i.e. if $M_i(l_i, v_i)$ is a proper k-set then $M_i'(l_i, v_i) = M_i(l_i, v_i)$. This result comes from a specific property of k-sets: the same computations on $KSET(k_1)$ and $KSET(k_2)$ with $k_1 \leq k_2$ either output the same results, or a pair made of $\top$ and a proper k-set.

(L3) It can be shown that if $M_s'(l_s, v_s) = \top$ then $M_n'(l_n, v_n) = \top$. Indeed, the same data dependencies hold in the (M_i)'s and in the (M_i')'s, and simple functions ensure that $\top$'s are systematically propagated in every instruction from input to output.

(L4) Finally, it is easy to prove that $M_s'(l_s, v_s) = \top$ because either $M_s(l_s, v_s)$ is an initial $\top$ value independent of the domain precision, or the predecessors of $M_s'(l_s, v_s)$ are the same proper k-sets than for $M_s(l_s, v_s)$ (by L1 and L2), hence $M_s'(l_s, v_s) = cast_{Kmax}(\{d_1, \ldots, d_q\})$ with $q > Kmax$, which evaluates to $\top$. The conclusion follows as explained previously.

In the general case, both the journal and guards are necessary to handle array expressions and dynamic jumps. The main problem is that $KSET(Kmax)$ propagation could compute more precise k-sets on array expressions / jump targets, involving less value propagation and breaking lemmas L2 and L3. Guards ensure that intermediate array expressions and jumps evaluate to proper k-sets in the (M_i)'s. Result L2 can still be proved, and intermediate array expressions and jumps have even the same values in (M_i)'s and (M_i')'s. L3 is also satisfied thanks to both lemma L2 and the restriction of backward propagation (through the journal) to effectively observed jump targets / array indexes. The rest of the proof remains the same.

5 Experiments

Implementation. Procedure PaR has been implemented into a prototype performing CFG reconstruction from 32-bit PowerPC (PPC) executable files (ELF format). The prototype is about 29 kloc of C++. The forward propagation follows a standard flow-sensitive scheme, enhanced with standard efficient data structures [21] to represent memory states. Procedures are inlined, calls and returns being identified syntactically. Array writes and dynamic jumps are guarded, while array reads are not. CFG reconstruction and instruction decoding are interleaved, i.e. the analysis can discover new instructions on-the-fly. This feature requires only slight modifications of the procedure and theoretical framework presented so far (cf. [19]) with no incidence on the main results. The tool can handle low-level features such as instruction overlapping and call stack overwriting, but self-modifying code is not addressed. Note that dealing with other instruction sets such as x86 is only a matter of front-end.

Goal. We are interested in evaluating the four following properties of PaR: (P1) absolute performance, both in terms of (a) precision and (b) efficiency, (P2) relative performance compared to standard approaches based on k-sets or intervals, (P3) locality of the refinement, (P4) scalability, and (P5) robustness to $Kmax$.

Test benches. Two test benches T1 and T2 are considered (cf. Table 1). T1 is a set of 12 small hand-written C programs compiled with `gcc`. They are intended to represent common situations where dynamic jumps arise from C code, like `switch` and function pointers. T2 is a single industrial case study (`aircraft`) taken from a safety-critical embedded software for aeronautics (developed by the SAGEM company). The functionalities of the program are not relevant for CFG reconstruction, details can be found in [2]. The source code size is about 21 kloc of C, compiled with `WindRiver`. The executable counts 32405 instructions and 51 dynamic jumps. All dynamic jumps have between 8 and 16 valid jump targets, for a total of 461 valid jump targets[1]. The program does not use dynamic memory allocation.

Protocol and results. To evaluate P1, we report: (a) the number of dynamic jumps for which precision requirements are met and the number of recovered invalid targets (∞ if a target evaluates to $\top$); (b) the computation time. To evaluate P2, PaR is compared to other straightforward flow-sensitive propagation procedures (with inlining): value analysis based on k-sets for different k, and global and undirected refinement of k-set propagation (start with k=1, when a target expression evaluates to $\top$, increase k by 5 - up to $Kmax$ - and restart the computation). Precision and efficiency are compared with the metrics of P1. We also perform a "mind experiment" (on T1 only) to compare the precision of PaR to the best possible precision of any convex domain-based analysis and strided interval-based analysis. To do so, results of PaR are compared to the best abstractions of each set of targets of each dynamic jump. Best abstractions are deduced through manual inspection. For P3, the minimal, maximal and average k-set sizes used by PaR are reported. T2 is used to evaluate P4. Finally, for P5 procedure PaR is launched with different $Kmax$ and we observe metrics of P1. Experiments were performed on a PC Intel 2Ghz equipped with 2 GBytes of RAM. Results are reported in Tables 2 to 4.

Comments. Results from Table 2 and Table 4 show that PaR is both reasonably efficient and very precise. Indeed, *no target expression evaluates to* $\top$ and very few false targets are recovered: 9 false targets for 123 valid targets on T1, 42 false targets for 461 valid targets on T2. Results from Table 3 and Table 4 shows that PaR exhibits very good locality : on each example, the maximal k-set employed is very close to the optimal one (max #T), and the average cardinality of k-sets is very low (< 1.2). Hence, precise computation is performed only on a small portion of the program. Moreover, as expected, the computation time is independent from $Kmax$ (Table 2 and Table 4) as long as $Kmax$ is large enough. These results also hold on T2, demonstrating the scalability of the approach. According to Table 2 and Table 4, PaR performs better than the other approaches considered here in at least precision, performance or robustness.

[1] This information was retrieved from `WindRiver` output for validation only, the prototype does not make any advantage of it.

	program	lines of C	#I	#DJ	#T	max#T
T1	test0	29	66	1	2	2
	test1	56	207	1	6	6
	test2	60	192	1	2	2
	test3	114	504	2	14	8
	test4	143	351	1	2	2
	test5	164	654	3	19	9
	test6	170	496	1	6	6
	test7	199	619	1	6	6
	test8	234	795	2	14	8
	test9	258	909	2	14	8
	test10	280	945	3	19	9
	test11	290	1006	3	19	9
T2	aircraft	21562	32405	51	461	16

#I: native code instructions
#DJ: dynamic jumps
#T: feasible targets
max#T: max. #T for a jump

Table 1. Test benches T1 and T2

	# SDJ	#FT	Time (s)
1-set propagation	0/21	∞/123	2
5-set propagation	6/21	∞/123	7
10-set propagation	13/21	∞/123	8
15-set propagation	21/21	9/123	12
50-set propagation	21/21	9/123	35
k-set iter ($Kmax = 50$)	21/21	9/123	29
PaR ($Kmax = 15$)	21/21	9/123	11
PaR ($Kmax = 50$)	21/21	9/123	11
PaR ($Kmax = 100$)	21/21	9/123	11
perfect Convex	21/21	5337/123	NA
perfect Strided Interval	21/21	498/123	NA

# SDJ: DJ with target $\neq \top$ w.r.t. total #DJ
# FT: recovered false targets w.r.t. total #T

Table 2. T1: Summarised results

program	max #T	min-k	max-k	avg-k
test0	2	0	3	1.08
test1	6	0	6	1.17
test2	2	0	3	1.18
test3	8	0	11	1.16
test4	2	0	3	1.12
test5	9	0	11	1.17
test6	6	0	6	1.11
test7	6	0	6	1.10
test8	8	0	12	1.12
test9	8	0	11	1.08
test10	9	0	13	1.12
test11	9	0	12	1.08

Table 3. T1: locality of PaR ($Kmax = 50$)

	#SDJ	#FT	max-k	avg-k	Time (mn)
1-set propagation	0/51	∞/461	NA	NA	1
5-set propagation	0/51	∞/461	NA	NA	7
10-set propagation	33/51	∞/461	NA	NA	24
15-set propagation	45/51	∞/461	NA	NA	26
20-set propagation	51/51	6/461	NA	NA	44
30-set propagation	51/51	6/461	NA	NA	44
50-set propagation	51/51	6/461	NA	NA	44
k-set iter ($Kmax = 50$)	51/51	6/461	NA	NA	103
PaR ($Kmax = 20$)	51/51	42/461	16	1.18	18
PaR ($Kmax = 30$)	51/51	42/461	16	1.18	18
PaR ($Kmax = 50$)	51/51	42/461	16	1.18	18

Table 4. Results on T2

The k-set propagation is very sensitive to $Kmax$, too coarse or/and too expensive when the bound is not well tuned. However, precision is very good when the bound is large enough. On T2, the recovery is slightly better than the one achieved by PaR, illustrating that the refinement process is not perfect. The naive k-set refinement method is also slightly more precise than PaR, but much more expensive. Finally, on these examples, PaR outperforms largely in terms of precision any value analysis based on convex domains (at least 5337 false targets) or strided intervals (at least 498 false targets).

These experiments show that PaR is a very precise and robust approach to CFG reconstruction with very moderate cost. PaR outperforms standard approaches in terms of preciseness (convex-based domains, k-set with small bound), cost (k-set with large bound or global refinement) or robustness (k-set with manual setting).

6 Related Work

The recent work of Thakur *et al.* [23] describes a refinement-based safety analysis on unstructured programs. A refined CFG reconstruction is performed, but the preciseness of the recovery is driven by the needs of the safety property to check, while the present work targets a precise recovery prior to any other analysis. Noticeably, the technique can cope with self-modifying code. CFA analysis and CFG reconstruction for structured programs have already been discussed in the introduction.

CFG reconstruction for unstructured programs. Industrial tools like IDA PRO [24] or AIT [12] usually rely on linear sweep decoding (brute force decoding of all code addresses) or recursive traversal (recursive decoding until a dynamic jump is encountered), enhanced with limited constant propagation, pattern matching techniques based on the knowledge of the compiling chain process and user annotations. These techniques are unsafe on general programs, missing many legal targets and branches (see experiments with IDA PRO in [4, 18]). The only safe techniques we are aware of are those by Reps *et al.* [5, 7, 8] and Kinder and Veith [18, 19], already presented in the introduction. None of these approaches rely on refinement. Note that the line of work presented in [5, 7, 8] addresses more issues than CFG reconstruction, like variable recovery and type reconstruction. Finally, some recent tools are able to generate test data from executable files [3, 4, 14], but they do not propose any safe CFG reconstruction.

Refinement-based software analysis. Since a decade, many refinement-based software verification techniques have been developed, either in static analysis [11, 13, 16, 17, 20] or in software model checking [6, 15]. In client-driven analysis [13], precision requirements are given by an external client and refinement is directed by data-dependencies. However, the precision of the analysis depends on the query to solve, precision is increased through adjusting the degree of flow/context-sensitiveness, and backward data-dependencies are not pruned by a journal mechanism. Other standard refinements [11, 16, 17, 20] include various kinds of "CFG refinements" (node splitting, trace partitioning) and enriching common domains with non-mergeable boolean values. These techniques are orthogonal to the domain refinement proposed here.

CEGAR [6, 10, 15] offers a systematic way of refining predicate domains through error trace analysis. However, it requires a very precise and expensive symbolic execution on abstract traces, as well as a clear and effective notion of (concrete) error trace, which is lacking for CFG recovery because of junk instructions.

7 Conclusion

This paper introduces the problem of Value Analysis with Precision Requirements (VAPR) and show how CFG reconstruction can be performed within VAPR. We then describe a refinement-based static analysis for VAPR, working over finite sets of constant values. Requirement propagation allows to automatically adjust the domain precision only where it is needed, resulting in a very precise CFG recovery at moderate cost. The procedure is sound, terminates and is relatively complete on a class of nontrivial programs. First experiments, including an industrial case study, show that the method does scale to realistic problems and outperforms standard analyses in terms of precision, efficiency or robustness. Future works comprise improving performance, conducting wider experiments and exploring other applications of the VAPR framework.

Acknowledgements. We are very grateful to Jérôme Leroux, Éric Goubault and the anonymous referees for their insightful comments, and to Philippe Baufreton and SAGEM for the case-study.

References

1. Balakrishnan, G., Gruian, R., Reps, T. W., Teitelbaum, T.: CodeSurfer/x86-A Platform for Analyzing x86 Executables. In: CC 2005. Springer, Heidelberg (2005)
2. Baufreton P., Heckmann, R.: Reliable and precise wcet and stack size determination for a real-life embedded application. In: ISoLA, Workshop On Leveraging Applications of Formal Methods, Verification and Validation, Poitiers-Futuroscope, France, December 12-14, 2007.
3. Bardin, S., Herrmann, P.: Structural Testing of Executables. In: IEEE ICST 2008. IEEE Computer Society, Los Alamitos (2008)
4. Bardin, S., Herrmann, P.: OSMOSE: Automatic Structural Testing of Executables. International Journal of Software Testing, Verification and Reliability (STVR). doi 10.1002/stvr.423
5. Balakrishnan, G., Reps, T. W.: Analyzing memory accesses in x86 executables. In: CC 2004. Springer, Heidelberg (2004)
6. Ball, T., Rajamani, S.: The SLAM project: Debugging system software via static analysis. In: POPL 2002. ACM, New York (2002)
7. Balakrishnan, G., Reps, T. W.: DIVINE: DIscovering Variables IN Executables. In: VMCAI 2007. Springer, Heidelberg (2007)
8. Balakrishnan, G., Reps, T. W.: Analyzing Stripped Device-Driver Executables. In: TACAS 2008. Springer, Heidelberg (2008)
9. Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: POPL 1977. ACM, New York (1977)
10. Clarke, E. M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counter Example-Guided Abstraction Refinement for Symbolic Model Checking. Journal of the ACM, 50(5), 2003
11. Dhurjati, D., Das, M., Yang, Y.: Path-Sensitive Dataflow Analysis with Iteration Refinement. In: SAS 2006. Springer, Heidelberg (2006)
12. Ferdinand, C., Heckmann, R.: aiT: worst case execution time prediction by static program analysis. In: IFIP Congress Topical Sessions 2004. Kluwer, Dordrecht (2004)
13. Guyer, S. Z., Lin, C.: Client-driven pointer analysis. In: SAS 2003. Springer, Heidelberg (2003)
14. Godefroid, P., Levin, M. Y., Molnar, D.: Automated Whitebox Fuzz Testing. In: NDSS 2008. The Internet Society (2008)
15. Henzinger, T. A., Jhala, R., Majumbar, R., Sutre, G.: Lazy Abstraction. In: POPL 2002. ACM, New York (2002)
16. Handjieva, M., Tzolovski, S.: Refining static analyses by trace-based partitioning using control-flow. In: SAS 1998. Springer, Heidelberg (1998)
17. Jeannet, B., Halbwachs, N., Raymond, P.: Dynamic partitioning in analyses of numerical properties. In: SAS 1999. Springer, Heidelberg (1999)
18. Kinder, J., Veith, H.: Jakstab: A Static Analysis Platform for Binaries. In: CAV 2008. Springer, Heidelberg (2008)
19. Kinder, J., Zuleger, F., Veith, H.: An Abstract Interpretation-Based Framework for Control Flow Reconstruction from Binaries. In: VMCAI 2008. Springer, Heidelberg (2009)
20. Mauborgne, L., Rival, X.: Trace Partitioning in Abstract Interpretation Based Static Analyzers . In: ESOP 2005. Springer, Heidelberg (2005)
21. Myers, E. W.: Efficient Applicative Data Types. In: POPL 1984. ACM, New York (1984)
22. Shivers, O.: Control-Flow Analysis in Scheme. In: PLDI 1988. ACM, New York (1988)
23. Thakur, A. V., Lim J., Lal, A., Burton, A., Driscoll, E., Elder, M., Andersen, T., Reps, T. W.: Directed Proof Generation for Machine Code. In: CAV 2010. Springer, Heidelberg (2010)
24. IDA Pro homepage `http://www.hex-rays.com/idapro`

A Details about the procedure PaR

PROPAGATE* : $(P : \mathcal{P}) \times (\mathcal{C} : \mathcal{R}) \times (M : \mathcal{M}) \times (D : \mathcal{D}) \mapsto \mathcal{M} \times \mathcal{H}_1 \times \mathcal{H}_2 \times \mathcal{S}$

A mostly standard forward (abstract) lfp computation procedure. The main difference are:

1. $\top_{\langle d_1,\ldots,d_q\rangle}$ and $\top_{init}$ propagated as simple $\top_*$, but kept in place (not overwritten by $\top_*$)
2. when $\{d_1,\ldots,d_q\}$ is abstracted to $\top$, it takes value $\top_{\langle d_1,\ldots,d_q\rangle}$
3. Operations are performed with maximal precision, then cast to the target precision
4. Faulty locations are systematically discarded (no propagation anymore) and recorded in S
5. The journal (H_1, H_2) is updated during propagation (see below)

——-

General scheme of the procedure:

```
S := ∅ ; (H_1, H_2) := empty-journal
While M not a fixpoint of P do
    forall instruction (l, T(l)) ∈ L × I of P do
        if for all φ⟨l, v⟩ ∈ C, M(l, v) ⊨ φ⟨l, v⟩ then
            M' := eval((l, i), M, D)        /* D is needed for cast between domains */
            (H_1, H_2) := update-journal((l, i), M, D, H_1, H_2)
            M := M ⊔ M'
        end if
    end forall
end while
S := {(l, v) ∈ L × V^# | M(l, v) ⊭ C}
return (M, H_1, H_2, S)
```

——-

journal updates : H_1 and H_2 are updated according to the following events:

- when instruction $(l, i) = (l, cgoto\ v)$ is fired, then $H_1(l, i, v) := M(l, v)$
- when $(l, i) = (l, ite(p, l_1, l_2))$ is fired for p evaluating to b, then $H_2(l, i, b) := true$
- when $(l, i) = (l, a[e] := ...)$ is fired, then $H_1(l, i, e) := M(l, e)$
- when $(l, i) = (l, ... := f(a[e], v_1, ..., v_n))$ is fired, then $H_1(l, i, e) := M(l, e)$

——-

cast : $(\widehat{d} : KSET(\infty)) \times (K : \mathbb{N}) \mapsto KSET(K)$

```
if |$\widehat{d}$| ≤ K then return $\widehat{d}$ else return ⊤ end if
```

——-

Propagation rules for $\top$**-labels**: we consider the merge of two $\top$-labels f_1, f_2, f_2 being the current value. Typically $f_2 := f_1 \sqcup_F f_2$. $f_\#$ denotes any $\top$-label value. Note that $\sqcup_F$ is not commutative.

$f_\# \sqcup_F f_{init} \mapsto f_{init}$ /* f_{init} *cannot be overwritten* */

$f_\# \sqcup_F f_{\langle d_1,\ldots,d_q\rangle} \mapsto f_{\langle d_1,\ldots,d_q\rangle}$ /* $f_{\langle\ldots\rangle}$ *cannot be overwritten* */

$f_{\{*,\bot\}} \sqcup_F f_* \mapsto f_*$

$f_* \sqcup_F f_\bot \mapsto f_*$

$f_\bot \sqcup_F f_\bot \mapsto f_\bot$

$f_{init} \sqcup_F f_{\{*,\bot\}} \mapsto f_*$ /* f_{init} *not propagated* */

$f_{\langle d_1,\ldots,d_q\rangle} \sqcup_F f_{\{*,\bot\}} \mapsto f_*$ /* $f_{\langle\ldots\rangle}$ *not propagated* */

Procedure 5: Detailed forward value propagation

```
**prop-req-expr** : $(e : \mathcal{E}) \times (H_1 : \mathcal{H}_1) \times (l : L) \times (i : I) \mapsto 2^{V^{\#}}$
  **case** $e$ **of**
    | $k \in \mathbb{N}$ : return $\emptyset$
    | $v \in V$ : return $\{v\}$
    | $a[k]$, $k \in \mathbb{N}$ : return $\{a[k]\}$
    | $op(e)$ : return **prop-req-expr**($e, H_1, l, i$)
    | $op(e_1, e_2)$ : return **prop-req-expr**($e_1, H_1, l, i$) $\cup$ **prop-req-expr**($e_2, H_1, l, i$)
    | $a[e]$ : **if** $H_1(l, i, e) = \top$ **then** return **prop-req-expr**($e, H_1, l, i$)
          **else** return $\{a[k] | k \in H_1(l, i, e)\}$
          **end if**
  **end case**
—-
**prop-req-instr** : $((l_0, i) : L \times I) \times ((l, v) : L \times V^{\#}) \times (M : \mathcal{M}) \times (H_1 : \mathcal{H}_1) \times (H_2 : \mathcal{H}_2) \mapsto \mathcal{S}$
  /* *backward propagation from faulty location* $(l, v)$ *for instruction* $(l_0, i)$ */
  **if** $M(l, v) = \top_{init}$ or $M(l, v) = \top_{\langle \dots \rangle}$ **then** return $\emptyset$
  **else if** $M(l, v) \neq \top_*$ **then** return $\emptyset$
  **else case** $l_0, i$ **of**
    | $l_0, goto\ l_1$ : **if** $l = l_1$ **then** return $\{(l_0, v)\}$ **else** return $\emptyset$
    | $l_0, cgoto(x)$: **if** $l \in H_1(l_0, i, x)$ return $\{(l_0, v)\}$ **else** return $\emptyset$
    | $l_0, ite(p, l_1, l_2)$ : **if** $l = l_1$ and $H_2(l_0, i, true) = true$ **then** return $\{(l_0, v)\}$
                  **else if** $l = l_2$ and $H_2(l_0, i, false) = true$ **then** return $\{(l_0, v)\}$
                  **else** return $\emptyset$
                  **end if**
    | $l_0, v' := e, l_1$ and $l_1 \neq l$ : return $\emptyset$
    | $l_0, v' := e, l$ :
          **if** $v = v'$ **then**
            return $\{l_0\} \times$**prop-req-expr**($e, H_1, l, i$)
          **else if** $v = a[k]$ and $v' = a[e']$ and $\{k\} = H_1(l_0, i, e')$ **then**
            return $\{l_0\} \times$**prop-req-expr**($e, H_1, l, i$)
          **else if** $v = a[k]$ and $v' = a[e']$ and $k \in H_1(l_0, i, e')$ **then**
            return ($\{l_0\} \times$**prop-req-expr**($e, H_1, l, i$)) $\bigcup$ $\{(l_0, v)\}$
          **else** return $(l_0, v)$
          **end if**
  **end case**
  **end if**
```

Procedure 6: Detailed backward requirement propagation

B Comparison of PaR and standard approaches

A summary of pros and cons of each approach for CFG reconstruction is depicted in Table 5. We consider precision of the recovery, computation time and robustness to initial parameters (essentially $Kmax$). It can be seen that PaR is superior to any of the other (standard) static analyses considered in Section 5: interval propagation and k-set propagation with small $Kmax$ are too imprecise, k-set propagation with large $Kmax$ and k-set propagation with refinement are very expensive (but slightly more precise). Moreover, k-set propagations without refinement are too much sensitive to the $Kmax$ parameter.

procedure	precision	efficiency	robustness
PaR	++	+	++
intervals	--	++	++$^{(*)}$
k-sets, low $Kmax$	--	++	-
k-sets, high $Kmax$	+++	--	-
k-set iter, global refinement	+++	--	++

(*) there is no parameter similar to $Kmax$

Table 5. Comparison between PaR and standard value analyses for CFG reconstruction