February 2019

Automatic Program Verification with SEAHORN

Arie GURFINKEL [a,1] and Jorge A. NAVAS [b]
[a] *Department of Electrical and Computer Engineering, University of Waterloo*
[b] *SRI International*

Abstract. In this paper, we present SEAHORN, a software verification framework. The key distinguishing feature of SEAHORN is its modular design that separates the concerns of the syntax of the programming language, its operational semantics, and the verification semantics. SEAHORN encompasses several novelties: it (a) encodes verification conditions using an efficient yet precise inter-procedural technique, (b) provides flexibility in the verification semantics to allow different levels of abstraction, (c) uses Horn-clauses as an intermediate language to represent verification conditions which simplifies interfacing with multiple verification tools based on Horn-clauses, and (d) leverages the state-of-the-art in software model checking and abstract interpretation for verification. SEAHORN provides users with a powerful verification tool and researchers with an extensible and customizable framework for experimenting with new software verification techniques.

Keywords. software Model Checking, program verification, Constrained Horn Clauses, Abstract Interpretation

1. Introduction

In this paper, we describe the SEAHORN verification framework. SEAHORN extends the LLVM [78] compiler infrastructure with verification techniques based on Software Model Checking and Abstract Interpretation. The framework provides many components that can be combined together for a variety of analysis needs. Many useful analyzers (e.g., memory safety, overflow checker, null pointer checker, etc.) are provided out of the box. The paper presents an overview of the framework and detailed description of the two verification engines: SPACER for Model Checking and CRAB for Abstract Interpretation.

In the rest of this section, we summarize the key unique features of the framework. First, SEAHORN decouples a programming language syntax and semantics from the underlying verification technique. Different programming languages include a diverse assortments of features, many of which are purely syntactic. Handling them fully is a major effort for new tool developers. We tackle this problem in SEAHORN by separating the language syntax, its operational semantics, and the underlying verification semantics – the semantics used by the verification engine. Specifically, we use the LLVM front-end(s) to deal with the idiosyncrasies of the syntax. We use LLVM intermediate repre-

[1] Corresponding Author: Arie Gurfinkel, Department of Electrical and Computer Engineering, University of Waterloo, Canada, E-mail: arie.gurfinkel@uwaterloo.ca.

sentation (IR), called the *bitcode*, to deal with the operational semantics, and apply a variety of transformations to simplify it further. In principle, since the bitcode has been formalized [102], this provides us with a well-defined formal semantics. Finally, we use Constrained Horn Clauses (CHC) to logically represent the verification conditions (VC).

Second, SEAHORN provides an efficient and precise analysis of programs with procedures using inter-procedural (i.e., modular) verification techniques. SEAHORN summarizes the input-output behavior of procedures efficiently without inlining. The expressiveness of the summaries is not limited to linear arithmetic, but extends to richer logics, including, for instance, arrays. Furthermore, SEAHORN includes a program transformation that lifts deep assertions closer to the main procedure. This increases context-sensitivity of intra-procedural analyses (used both in verification and compiler optimization), and has a significant impact on our inter-procedural verification algorithms.

Third, SEAHORN allows developers to customize the verification semantics and offers users with verification semantics of various degrees of abstraction. SEAHORN is fully parametric in the (small-step operational) semantics used for the generation of VCs. The level of abstraction in the built-in semantics varies from considering only LLVM numeric registers to considering the whole heap (modeled as a collection of non-overlapping arrays). In addition to generating VCs based on small-step semantics [90], SEAHORN can also automatically lift small-step semantics to large-step [6, 57] (a.k.a. Large Block Encoding, or LBE).

Fourth, SEAHORN uses Constrained Horn Clauses (CHC) as its intermediate verification language. CHC provide a convenient and elegant way to formally represent many encoding styles of verification conditions. The recent popularity of CHC as an intermediate language for verification engines makes it possible to interface SEAHORN with a variety of new and emerging tools.

Fifth, SEAHORN builds on the state-of-the-art in Software Model Checking (SMC) and Abstract Interpretation (AI). SMC and AI have independently led over the years to the production of analysis tools that have a substantial impact on the development of real world software. Interestingly, the two exhibit complementary strengths and weaknesses (see e.g., [1, 9, 46, 56]). While SMC so far has been proved stronger on software that is mostly control driven, AI is quite effective on data-dependent programs. SEAHORN combines SMT-based model checking techniques with program invariants supplied by an Abstract Interpreter.

Finally, SEAHORN is open-sourced and is implemented on top of the open-source LLVM compiler infrastructure. LLVM is a well-maintained, well-documented, and continuously improving framework. This allows SEAHORN users to easily integrate program analyses, transformations, and other tools that targets LLVM. Moreover, since SEAHORN analyses LLVM IR, this allows to exploit a rapidly-growing frontier of LLVM front-ends, encompassing a diverse set of languages. SEAHORN itself is released as open-source as well (source code can be downloaded from `http://seahorn.github.io`).

The design of SEAHORN provides users, developers, and researchers with an extensible and customizable environment for experimenting with and implementing new software verification techniques. SEAHORN is implemented in C++ in the LLVM compiler infrastructure [78]. The overall approach is illustrated in Figure 1. SEAHORN has been developed in a modular fashion; its architecture is layered in three parts:

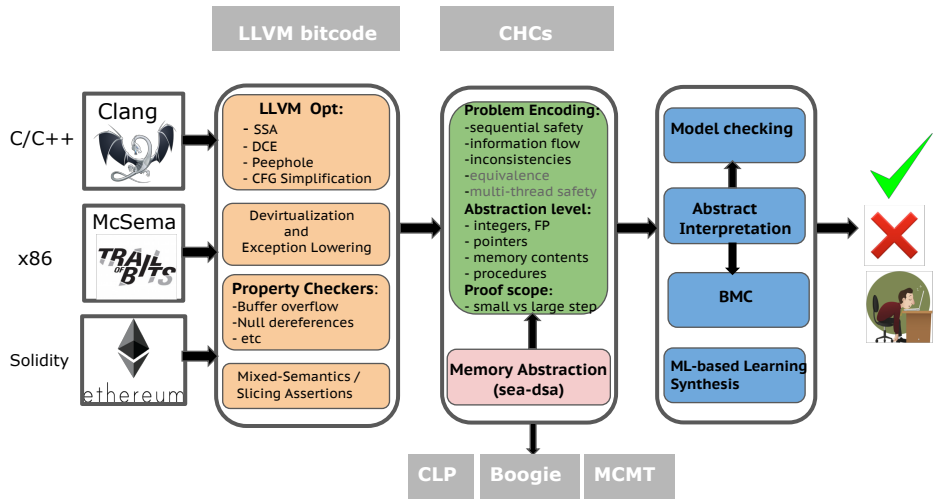


Figure 1. Overview of SEAHORN architecture.

Front-End: Takes an LLVM based program (e.g., C) input program and generates LLVM IR bitcode. Specifically, it performs the pre-processing and optimization of the bitcode for verification purposes. More details are reported in Section 2.

Middle-End: Takes as input the optimized LLVM bitcode and emits verification condition as Constrained Horn Clauses (CHC). The middle-end is in charge of selecting the encoding of the VCs and the degree of abstraction. VCs can be exported to different formats such as Constraint Logic Programming (CLP), Boogie or MCMT (Model Checking Modulo Theories). More details are reported in Section 3.

Back-End: Takes CHC as input and outputs the result of the analysis. In principle, any verification engine that digests CHC clauses could be used to discharge the VCs. Currently, SEAHORN employs an SMT-based model checking engine SPACER [74]. Complementary, SEAHORN uses the abstract interpretation-based analyzer CRAB for providing numerical invariants. More details are reported in Section 4.

Related Work. Automated analysis of software is an active area of research. There is a large number of tools with different capabilities and trade-offs [5, 7, 8, 18, 20–22, 32, 82]. Our approach on separating the program semantics from the verification engine has been previously proposed in numerous tools. From those, the tool SMACK [91] is the closest to SEAHORN. Like SEAHORN, SMACK targets programs at the LLVM-IR level. However, SMACK targets Boogie intermediate verification language [35] and Boogie-based verifiers to construct and discharge the proof obligations. SEAHORN differs from SMACK in several ways: (i) SEAHORN uses CHC as its intermediate verification language, which allows to target different solvers and verification techniques (ii) it tightly integrates and combines both state-of-the-art software model checking techniques and abstract interpretation and (iii) it provides an automatic inter-procedural analysis to reason modularly about programs with procedures.

Inter-procedural and modular analysis is critical for scaling verification tools and has been addressed by many researchers (e.g., [2, 67, 74, 77, 80, 96]). Our approach of using mixed-semantics [63] as a source-to-source transformation has been also explored in [77]. While in [77], the mixed-semantics is done at the verification semantics (Boogie in this case), in SEAHORN it is done in the front-end level allowing mixed-semantics to interact with compiler optimizations.

Constrained Horn clauses have been proposed [12] as an intermediate (or exchange) format for representing verification conditions. However, they have long been used in the context of static analysis of imperative and object-oriented languages (e.g., [81, 90]) and more recently adopted by an increasing number of solvers (e.g., [13, 42, 67, 74, 80]) as well as other verifiers such as UFO [3], HSF [53], VeriMAP [33], Eldarica [93], and TRACER [68].

A previous version of this paper has appeared in [58].

2. Pre-processing for Verification

In our experience, performance of even the most advanced verification algorithms is significantly impacted by the front-end transformations. In SEAHORN, the front-end plays a very significant role in the overall architecture.

In principle, SEAHORN can take any input program that can be translated into LLVM bitcode. However, SEAHORN has been highly customized to analyze C programs as translated by `clang`, and thus, analysis of C code is the most prominent SEAHORN's strength. More recently, SEAHORN has increasingly supported analysis of C++ programs, and we plan to continue doing that.

Our goal is to make SEAHORN not to be limited to C or C++ programs, but applicable (with various degrees of success) to a broader set of languages based on LLVM (e.g., Objective C, Rust, and Swift). For instance, we have recently added support for two very different languages: x86 binary programs using the McSema [98] tool, and Solidity smart contracts using a just-in-time compiler for Ethereum EVM code [37]. Although the verification results have been relatively modest compared with verification of C programs, they demonstrate the broad applicability of SEAHORN.

Once we have obtained LLVM bitcode, the front-end is split into two main subcomponents. The first one is a pre-processor that performs optimizations and transformations. This pre-processing is largely optional. Its main goal is to transform the LLVM bitcode to make the verification task *easier*. The second part is focused on a reduced set of transformations mostly required to produce correct results even if the pre-processor is disabled. It also performs SSA transformation and internalizes functions, but in addition, lowers `switch` instructions into `if-then-else`s, ensures only one exit block per function, inlines global initializers into the main procedure, and identifies `assert`-like functions.

The front-end can optionally inline functions. This is often useful to simplify verification tasks, and is also necessary for precise Bounded Model Checking (and, currently, is required for counterexample generation).

One typical problem in proving safety of large programs is that assertions can be nested very deep inside the call graph. As a result, counterexamples are longer and it is harder to decide for the verification engine what is relevant for the property of interest.

main ()
  $p1$ (); $p1$ ();
  assert ($c1$);
$p1$ ()
  $p2$ ();
  assert ($c2$);
$p2$ ()
  assert ($c3$);
	$main_{new}$ ()
	  if (*) goto $p1_{entry}$;
	  else $p1_{new}$ ();
	  if (*) goto $p1_{entry}$;
	  else $p1_{new}$ ();
	  if ($\neg c1$) goto *error*;
	  assume (false);
		$p1_{entry}$:
		  if (*) goto $p2_{entry}$;
		  else $p2_{new}$ ();
		  if ($\neg c2$) goto *error*;
		$p2_{entry}$:
		  if ($\neg c3$) goto *error*;
		  assume (false);
		error : assert (false);
			$p1_{new}$ ()
			  $p2_{new}$ ();
			  assume ($c2$);
			$p2_{new}$ ()
			  assume ($c3$);

Figure 2. A program before (left) and after (right) mixed-semantics transformation.

To mitigate this problem, the front-end provides a transformation based on the concept of *mixed-semantics*[2] [63,77]. It relies on the simple observation that any call to a procedure P either fails inside the call and therefore P does not return, or returns successfully from the call. Based on this, any call to P can be instrumented as follows:

- if P may fail, then make a copy of P's body (in main) and jump to the copy.
- if P may succeed, then make the call to P as usual. Since P is known not to fail each assertion in P can be safely replaced with an *assume*.

Upon completion, only the main function has assertions and each procedure is inlined at most once. The explanation for the latter is that a function call is inlined only if it fails and hence, its call stack can be ignored. Mixed-semantics transformation preserves reachability and non-termination properties [63]. Since this transformation is not very common in other verifiers, we illustrate it on an example.

Example 1 (Mixed-semantics transformation) *On the left in Figure 2 we show a small program consisting of a main procedure calling two other procedures $p1$ and $p2$ with three assertions $c1$, $c2$, and $c3$. On the right, we show the new program after the mixed-semantics transformation. First, when main calls $p1$ it is transformed into a non-deterministic choice between (a) jumping into the entry block of $p1$ or (b) calling $p1$. The case (a) represents the situation when $p1$ fails and it is done by inlining the body of $p1$ (labeled by $p1_{entry}$) into main and adding a* goto *statement to $p1_{entry}$. The case (b) considers the case when $p1$ succeeds and hence it simply duplicates the function $p1$ but replacing all the assertions with assumptions since no failure is possible. Note that while $p1$ is called twice, it is inlined only once. Furthermore, each inlined function ends up with an "*assume *(false)" indicating that execution dies. Hence, any complete execution of a transformed program corresponds to a bad execution of the original one. Finally, an interesting side-effect of mixed-semantics is that it can provide some context-sensitivity to context-insensitive intra-procedural analyses.*

3. Verification Conditions

SEAHORN provides out-of-the-box verification semantics with different degrees of abstraction. Furthermore, to accommodate a variety of applications, SEAHORN is designed

[2]The semantics is called *mixed* because it combines small- and big-step operational semantics.

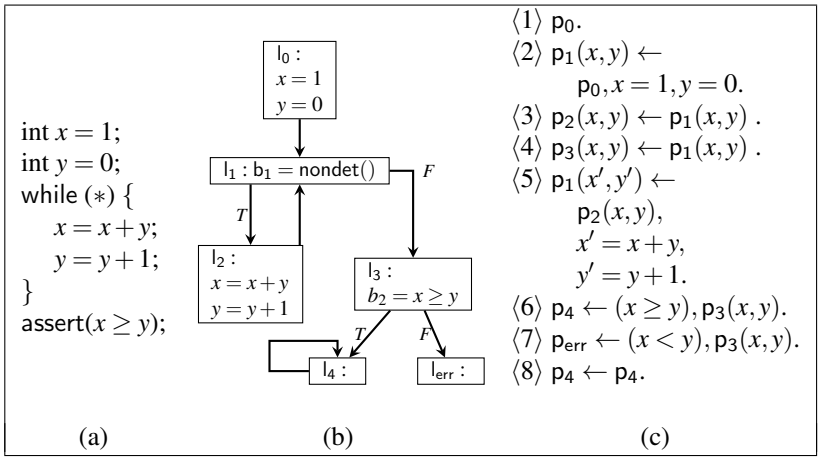


Figure 3. (a) Program, (b) Control-Flow Graph, and (c) Verification Conditions.

to be easily extended with a custom semantics as well. In this section, we illustrate the various dimensions of semantic flexibility present in SEAHORN.

Encoding Verification Conditions. SEAHORN is parametric in the semantics used for VC encoding. It provides two different semantics encodings: (a) a small-step encoding (exemplified in Figure 3) and (b) a large-step encoding. Large-step encoding is similar to the Large Block Encoding (LBE) of [6]. A user can choose the encoding depending on the particular application. In practice, large-step is often more efficient but small-step might be more useful if a fine-grained proof or counterexample is needed. For example, SEAHORN used the large-step encoding in SV-COMP [59].

Regardless of the encoding, SEAHORN uses Constrained Horn Clauses (CHC) to encode the VCs. Given the sets $\mathscr{F}$ of function symbols, $\mathscr{P}$ of predicate symbols, and $\mathscr{V}$ of variables, a *Constrained Horn Clause (CHC)* is a formula in First Order Logic of the following form:

$$\forall \mathscr{V} \cdot (\phi \wedge p_1[X_1] \wedge \cdots \wedge p_k[X_k] \rightarrow h[X]), \text{ for } k \geq 0$$

where ϕ is a constraint over $\mathscr{F}$ and $\mathscr{V}$ with respect to some background theory T; $X_i, X \subseteq \mathscr{V}$ are (possibly empty) vectors of variables; $p_i[X_i]$ is an application $p(t_1,\ldots,t_n)$ of an n-ary predicate symbol $p \in \mathscr{P}$ for first-order terms t_i constructed from $\mathscr{F}$ and X_i; and $h[X]$ is either defined analogously to p_i or is $\mathscr{P}$-free (i.e., no $\mathscr{P}$ symbols occur in h). Here, h is called the *head* of the clause and $\phi \wedge p_1[X_1] \wedge \ldots \wedge p_k[X_k]$ is called the *body*. A clause is called a *query* if its head is $\mathscr{P}$-free, and otherwise, it is called a *rule*. A rule with body true is called a *fact*. We say a clause is *linear* if its body contains at most one predicate symbol, otherwise, it is called *non-linear*. In this paper, we follow the Constraint Logic Programming (CLP) convention of representing Horn clauses as $h[X] \leftarrow \phi, p_1[X_1], \ldots, p_k[X_k]$, by omitting explicit universal quantification, replacing implication by an arrow and conjunction by a comma, and writing clauses as rules with the head on the left.

A set of CHCs is satisfiable if there exists a First Order Logic interpretation $\mathscr{I}$ of the predicate symbols $\mathscr{P}$ such that each constraint ϕ is true under $\mathscr{I}$. Without loss of

generality, deciding whether a program $\mathscr{A}$ satisfies a safety property α_{safe} is reducible to establishing the (un)satifiability of CHCs encoding the VCs of $\mathscr{A}$. We illustrate the process on the following example. Additional examples of the encoding are available in [10].

Example 2 (Small-step encoding of VCs using CHCs) *Figure 3(a) shows a program that increments two variables x and y in a non-deterministic loop. After the loop is executed we would like to prove that x cannot be less than y. Ignoring overflow , it is easy to see that the program is safe since x and y are initially non-negative numbers and x is greater than y. Since the loop increases x by a greater amount than y, at its exit x cannot be smaller than y. Figure 3(b) depicts, its corresponding Control Flow Graph (CFG) and Figure 3(c) shows its VCs encoded as a set of CHCs.*

The set of CHCs in Figure 3(c) essentially represents the small-step *operational semantics of the CFG. Each basic block is encoded as a CHC. A basic block label l_i in the CFG is translated into a predicate $\mathsf{p_i}(X_1,\ldots,X_n)$ such that $\mathsf{p_i} \in \mathscr{P}$ and $\{X_1,\ldots,X_n\} \subseteq \mathscr{V}$ is the set of live variables at the entry of block l_i. A CHC can model both the control flow and data of each block in a succinct way. For instance, the fact $\langle 1 \rangle$ represents that the entry block l_0 is reachable. Clause $\langle 2 \rangle$ describes that if l_0 is reachable then l_1 should be reachable too. Moreover, its body contains the constraints $x = 1 \wedge y = 0$ representing the initial state of the program. Clause $\langle 5 \rangle$ models the loop body by stating that the control flow moves to l_2 from l_1 after transforming the state of the program variables through the constraints $x' = x + y$ and $y' = y + 1$, where the primed versions represent the values of the variables after the execution of the arithmetic operations. Based on this encoding, the program in Figure 3(a) is safe if and only if the set of recursive clauses in Figure 3(c) augmented with the query $\mathsf{p_{err}}$ is unsatisfiable. Note that since we are only concerned about proving safety (and not termination) any safe final state can be represented by an infinite loop (e.g., clause $\langle 8 \rangle$)).*

SEAHORN middle-end offers a very simple interface for developers to implement an encoding of the verification semantics that fits their needs. At the core of the SEAHORN middle-end lies the concept of a symbolic store. A *symbolic store* simply maps program variables to symbolic values. The other fundamental concept is how different parts of a program are *symbolically executed*. The small-step verification semantics is provided by implementing a symbolic execution interface that symbolically executes LLVM instructions relative to the symbolic store. This interface is automatically lifted to large-step semantics as necessary.

Modeling statements with different degrees of abstraction. The SEAHORN middle-end includes verification semantics with different levels of abstraction. Those are, from the coarsest to the finest:

Registers only: only models LLVM numeric registers. In this case, the constraints part of CHC is over the theory of Linear Integer Arithmetic (LIA).

Registers + Pointers (without memory content): models numeric and pointer registers. This is sufficient to capture pointer arithmetic and determine whether a pointer is `NULL`. Memory addresses are also encoded as integers. Hence, the constraints remain over LIA.

Registers + Pointers + Memory: models numeric and pointer registers and the heap. The heap is modeled by a collection of non-overlapping arrays. The constraints are over the combined theories of arrays and LIA.

Memory encoding. For concrete semantics, SEAHORN assumes object-based memory model. Each allocation site (heap, stack, and globals) returns a memory object representing a sequence of bytes. The objects are disjoint. Each pointer points to some object and pointer arithmetic is restricted to stay within an object.

For verification condition, an abstract memory model is used that restricts the number of memory objects to be finite. As usual, an abstract memory object represents all concrete objects allocated at a given syntactic allocation site. The memory is further partitioned into regions, where a region is one or more memory objects, such that each memory instruction uses or modifies exactly one memory region. The abstract memory model is context-sensitive – each procedure has its own memory regions.

Memory regions are computed statically using a specialized context-sensitive alias analysis call SEADSA [60]. As the name suggests, SEADSA is a variant of *Data Structure Analysis (DSA)* [79]. DSA itself is an extension of Steensgaard's (a.k.a. unification-based) pointer analysis [97].

In SEADSA, the memory is partitioned into a heap, a stack, and global objects. The analysis builds for each function a *DS graph* where each node represents an abstract memory region. Distinct *nodes* express disjoint sets of memory objects. Edges in the graph represent *points-to* relationships between nodes. Each node is typed and determines the number of fields and outgoing edges in a node. A node can have one outgoing edge per field, but each field can have at most one outgoing edge. This restriction is key to scalability and it is preserved by *unifying* nodes whenever it is violated.

Given a DS graph, each node is mapped to an array in the VC. Then, each memory read (`load`) and write (`store`) in LLVM bitcode is associated with a unique node (i.e., the array). For memory writes, SEAHORN creates a new array variable representing the new state of the array after the write operation.

Inter-procedural proofs. For most real programs verifying a function separately from each possible caller (i.e., *context-sensitivity*) is necessary for scalability. The version of SEAHORN for SV-COMP 2015 [59] achieved full context-sensitivity by inlining all program functions. Although inlining is often an effective solution for small and medium-size programs it is well known that suffers from an exponential blow up in the size of the original program. Even more importantly, inlining cannot produce inter-procedural proofs nor counterexamples which are often highly desired.

We tackled this problem in [58], by providing an encoding that allows inter-procedural proofs. We illustrate this procedure via an example in Figure 4. The upper box shows a program with three procedures: *main*, *foo*, and *bar*. The program swaps two numbers x and y. The procedure *foo* adds two numbers and *bar* subtracts them. At the exit of *main* we want to prove that the program indeed swaps the two inputs. To show all relevant aspects of the inter-procedural encoding we add a trivial assertion in *bar* that checks that whenever x and y are non-negative the input x is greater or equal than the return value.

The lower box of Figure 4 illustrates the corresponding verification conditions encoded as CHCs. The new encoding follows a small-step style as the intra-procedural encoding shown in Figure 3 but with two major distinctions. First, notice that the CHCs

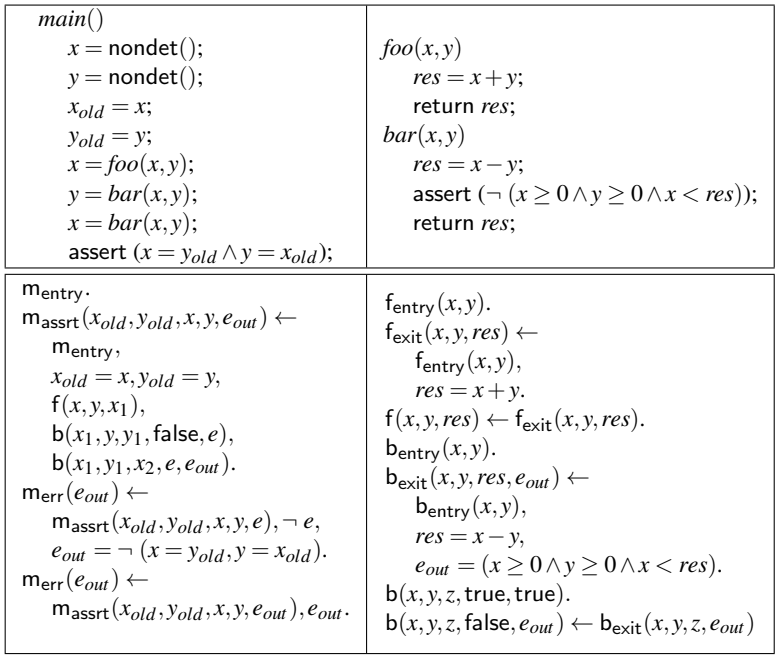


Figure 4. A program with procedures (upper) and its verification condition (lower).

are not linear anymore (e.g., the rule denoted by $\mathsf{m}_{\mathsf{assrt}}$). Each function call has been replaced with a *summary rule* (f and b) representing the effect of calling to the functions foo and *bar*, respectively. The second difference is how assertions are encoded. In the intra-procedural case, a program is unsafe if the query $\mathsf{p}_{\mathsf{err}}$ is satisfiable, where $\mathsf{p}_{\mathsf{err}}$ is the head of a CHC associated with a special basic block to which all can-fail blocks are redirected. However, with the presence of procedures assertions can be located deeply in the call graph of the program, and therefore, we need to modify the CHCs to propagate error to the main procedure.

In our example, since a call to *bar* can fail we add two arguments e_{in} and e_{out} to the predicate b where e_{in} indicates if there is an error before the function is called and e_{out} indicates whether the execution of *bar* produces an error. By doing this, we are able to propagate the error in clause $\mathsf{m}_{\mathsf{assrt}}$ across the two calls to *bar*. We indicate that no error is possible at *main* before any function is called by unifying false with e_{in} in the first occurrence of b. Within a can-fail procedure we skip the body and set e_{out} to true as soon as an assertion can be violated. Furthermore, if a function is called and e_{in} is already true we can skip its body (e.g., first clause of b). Functions that cannot fail (e.g., foo) are unchanged. The above program is safe if and only if the query $\mathsf{m}_{\mathsf{err}}(\mathsf{true})$ is unsatisfiable.

Finally, it is worth mentioning that this propagation of error is not required if the mixed-semantics transformation described in Section 2 is applied.

4. Verification Engines

In principle, SEAHORN can be used with any Horn clause or LLVM-based verification tool. In the following, we describe two such tools developed by ourselves. Notably,

the tools discussed below are based on the contrasting techniques of SMT-based model checking and Abstract Interpretation, showcasing the wide applicability of SEAHORN.

4.1. SMT-Based Model Checking with SPACER

SPACER is an efficient SMT-based Model Checker for deciding satisfiability of Constrained Horn Clauses (CHC) [73–75]. Of course, since CHC satisfiability is undecidable, we use the term *decision procedure* informally. SPACER is a sound procedure, but it is not complete (i.e., not formally a decision procedure, and does not terminate on all inputs). In contrast to other SMT-based Model Checking algorithms (for example, those based on based on interpolation [2,53,66,80]), the reasoning in SPACER is compositional (or modular). That is, the *transition relation* is not unrolled. SPACER reasons about a body of individual procedure (or predicate) at a time, and communicates information between procedures (or predicates) using summaries. This is crucial for scaling SMT-based Model Checking to programs. Unlike hardware circuits, an unrolling of a program (i.e., unrolling loops and inlining procedures) increases the size of an SMT formula representing a verification condition (VC) exponentially. The approach taken by SPACER avoids the exponential explosion by limiting the information that can be exchanged between procedures to well-defined summaries. The summaries also provide a form of caching to prevent exploring the same procedure in the same calling context multiple times.

SPACER is integrated into SMT-solver Z3 [34] and is currently the default CHC engine in Z3. It supports CHC with constraints in the (combined) theories of Linear Real Arithmetic [9], Linear Integer Arithmetic [75], Arrays [73], with basic support for theories of Bit Vectors and Abstract Data Types. Both quantifier free and universally quantified solutions for the theory of arrays are supported [61].

In this section, we give a high-level overview of SPACER algorithm. Many implementation details and optimizations are omitted since they often change between different versions of the implementation. SPACER builds on three main concepts: Craig Interpolation, Model Based Projection, and an IC3/PDR-style Model Checking algorithm. In the rest of this section, we describe each component in turn, starting with interpolation.

4.1.1. Craig Interpolation

Let A and B be two formulas in First Order Logic such that $A \wedge B$ is unsatisfiable. A *Craig interpolant* (or simply an interpolant) is a formula I such that

$$A \implies I \qquad\qquad I \implies \neg B$$

and the only uninterpreted constants and functions in I are those that are shared between A and B. For example, consider the following two formulas A and B in Linear Integer Arithmetic:

$$A = (a < x \wedge x < b) \qquad\qquad B = (a = 1 \wedge b = 1)$$

The conjunction $A \wedge B$ is unsatisfiable: if $a = b = 1$ then there cannot be an integer x strictly between a and b. The shared uninterpreted constants are a and b. There is an interpolant, but it is not unique. Several possible interpolants are:

$$I_1 = (a < b) \qquad I_2 = \neg(a = 1 \wedge b = 1) \qquad I_3 = (a \neq b)$$

All of the above formulas I_i (for $1 \leq i \leq 3$) is an interpolant. We write $\text{ITP}(A,B)$ for some interpolant between A and B if it exists.

It is well known that interpolants can be computed directly from a resolution refutation of satisfiability of A and B. In case of SMT, interpolation is a combination of interpolation over propositional resolution [62] and special procedures for theory lemmas and their derivation [19,54]. We refer the reader to the references above for more details.

In the case of SPACER, the interpolation problem is more restricted and simplified. We are only interested in computing an interpolant $\text{ITP}(A,B)$ in the case where B is a conjunction of literals and every uninterpreted symbol of B is shared with A. In this case, the simplest choice for an $\text{ITP}(A,B)$ is $\neg B$. Since $A \wedge B$ is unsatisfiable, it follows that $A \implies \neg B$, and obviously $\neg B$ implies itself. Note that in the example above, I_2 is simply $\neg B$. On one hand, using $\neg B$ as an interpolant defeats the purpose of interpolation. On the other, it provides a default case, that is often avoided, but is possible when a different interpolant is hard to compute. This is especially convenient for a system like Z3 that does not consistently produce easy-to-interpolate proofs.

Under the restrictions above, another alternative for an interpolant a negation of a Minimal Unsatisfiable Subset (MUS) of B. The reasoning is the same as for using $\neg B$. Such an interpolant does not do much generalization, but might filter irrelevant facts.

In practice, interpolant computation used by SPACER is a mix of proof-based and MUS-based procedure. A proof, and, in particular, theory lemmas of the proof, are examined to extract the interaction of B literals with the refutation. If the interaction is fairly clear, an *interpolating unsat core* is extracted by using interpolation-style reasoning. If the interaction is not clear, an MUS for B is computed. This style of reasoning enables SPACER to construct an interpolants such as I_1, I_2, and I_3 in the example above. However, the exact interpolant constructed depends on the proof produced by Z3.

4.1.2. Model-Based Projection

Let φ be a satisfiable formula with uninterpreted constants (or variables) $\mathit{Vars}(\varphi)$. Let U be a subset of variables in $\mathit{Vars}(\varphi)$, and $M \models \varphi$ be a model of φ. A formula ψ is a *Model Based Projection* (MBP) of U relative to M iff (a) $M \models \psi$, (b) $\psi \implies \exists U \cdot \varphi$, (c) $\mathit{Vars}(\psi) \subseteq \mathit{Vars}(\varphi) \setminus U$, and (d) ψ is a monomial (i.e., a conjunction of literals). Intuitively, MBP under-approximates projection (or quantifier elimination). Alternatively, MBP ψ can be seen as a generalization of the model M: ψ *contains* the model, yet, it is a formula (i.e., has a finite representation) and is contained in the projection $\exists U \cdot \varphi$.

We write $\text{MBP}(\exists U \cdot \varphi, M)$ for an MBP procedure that given an existentially quantified formula and a model, returns a corresponding model-based projection. An MBP procedure is *finite* if it is finite in the model argument. That is, the function $\lambda x \cdot \text{MBP}(\exists U \cdot \varphi, x)$ has a finite range. It is not difficult to see that a theory that admits quantifier elimination has a finite MBP. Consider a formula $\exists U \cdot \varphi$. Assume that there is an equivalent quantifier free formula ψ:

$$\psi \iff \exists U \cdot \varphi$$

Let $\psi_1 \vee \cdots \vee \psi_n$ be a DNF decomposition of ψ. Then, define $\text{MBP}(\exists U \cdot \varphi, M) = \psi_i$ such that $i = \min\{1 \leq j \leq n \mid M \models \psi_j\}$.

Conversely, a finite MBP provides a procedure for quantifier elimination. Let M_1 be a model for φ, and $\psi_1 = \text{MBP}(\exists U \cdot \varphi, M_1)$. Let M_2 be a model for $\varphi \wedge \neg\psi_1$, and $\psi_2 = \text{MBP}(\exists U \cdot \varphi, M_2)$, etc. Since by assumption MBP is finite, the number of such ψ_i is finite as well. Hence the formula $\psi_1 \vee \cdots \vee \psi_n$ is well defined, quantifier free, and is equivalent to $\exists U \cdot \varphi$.

A finite MBP for Linear Real Arithmetic has been introduced in [74]. Unlike quantifier elimination for LRA, it can be computed in linear time (assuming the model is given and evaluating literals in the model is constant time). A finite MBP for Linear Integer Arithmetic and Abstract Data Types has been presented in [11]. MBP for the theory of Arrays has been developed in [73]. Obviously, MBP for arrays is not finite.

We illustrate an MBP procedure for the combined theories of arrays and arithmetic using an example below. Let φ denote the formula

$$(b = a[i_1 \leftarrow v_1]) \vee (a[i_2 \leftarrow v_2][i_3] > 5 \wedge a[i_4] > 0)$$

where a and b are array variables whose index and value sorts are both Int, the sort of integers, and all other variables have sort Int. Here, for an array a, we use $a[i \leftarrow v]$ to denote a *store* of v into a at index i and use $a[i]$ to denote the value of a at index i. Suppose that we want to existentially quantify the array variable a. Let $M \models \varphi$. We will consider two possibilities for M:

1. Let $M \models b = a[i_1 \leftarrow v_1]$, i.e., M satisfies the array equality containing a. In this case, our MBP procedure substitutes the term $b[i_1 \leftarrow x]$ for a in φ, where x is a fresh variable of sort Int. That is, the result of MBP is $\exists x \cdot \varphi[b[i_1 \leftarrow x]/a]$.
2. Let $M \models b \neq a[i_1 \leftarrow v_1]$. We use the second disjunct of φ for MBP. Furthermore, let $M \models i_2 \neq i_3$. We then reduce the term $a[i_2 \leftarrow v_2][i_3]$ to $a[i_3]$ to obtain $a[i_3] > 5 \wedge a[i_4] > 0$, using the relevant disjunct of the select-after-store axiom of ARR. We then introduce fresh variables x_3 and x_4 to denote the two select terms on a, obtaining $x_3 > 5 \wedge x_4 > 0$. Finally, we add $i_3 = i_4 \wedge x_3 = x_4$ if $M \models i_3 = i_4$ and add $i_3 \neq i_4$ otherwise, choosing the relevant case of Ackermann reduction, and existentially quantify x_3 and x_4.

Model-Based Projection is crucial for SPACER. It is used both in computing predecessors and summaries. However, since its inception, it has found many other applications as well. For example, in [11] it is used in a procedure for deciding satisfiability of quantified formulas. In [40] to discover a simulation relation between different version of a program. In [72] it is extended with Skolemization and is used to synthesize implementation from assume-guarantee contracts. The Skolemization procedure is further improved in [39].

4.1.3. SPACER *Algorithm*

Without loss of generality, we assume that set of CHCs encoding safety of procedural programs is transformed into an equisatisfiable set of just *three* clauses with a *single* predicate symbol of the following form:

$$\begin{gathered} Inv(\bar{x}) \leftarrow Init(\bar{x}) \qquad \neg Bad(\bar{x}) \leftarrow Inv(\bar{x}) \\ Inv(\bar{x}') \leftarrow Inv(\bar{x}), Inv(\bar{x}^o), Tr(\bar{x}, \bar{x}^o, \bar{x}') \end{gathered} \tag{1}$$

The notation $\bar{x}^{\dagger}$ stands for a vector of variables obtained from $\bar{x}$ by adding † to every variable, where $\dagger \in \{', ^{o}\}$. For example, $(x_1, x_2, x_3)'$ is (x'_1, x'_2, x'_3). Intuitively, *Inv* is the program invariant, $\bar{x}$ denotes the pre-state of a program transition, $\bar{x}'$ denotes the post-state, and $\bar{x}^{o}$ denotes the summary of a procedure call (if one is made). Any verification condition for sequential programs can be transformed into the form of (1) by adding extra state variables that denote active program location and active procedure being executed.

In the special case of verification conditions of procedure-free sequential programs, $\bar{x}^{o}$ variables do not appear in *Tr* and the conjunct $Inv(\bar{x}^{o})$ can be dropped. The resulting three clauses simplify to the following:

$$\begin{array}{c} Inv(\bar{x}) \leftarrow Init(\bar{x}) \qquad \neg Bad(\bar{x}) \leftarrow Inv(\bar{x}) \\ Inv(\bar{x}') \leftarrow Inv(\bar{x}), Tr(\bar{x}, \bar{x}') \end{array} \tag{2}$$

In the case of (2), *Inv* denotes a regular inductive invariant of a transition system.

To simplify the notation, we introduce a special function $\mathscr{F}_{Tr}$, called a *forward transformer*, that replaces *Inv* in the rule by a pair of formulas $\phi_A(\bar{x})$ and $\phi_B(\bar{x})$. Formally, it is defined as follows:

$$\mathscr{F}_{Tr}(\varphi_A, \varphi_B) \quad \equiv \quad Init(\bar{x}') \vee \left(\varphi_A(\bar{x}) \wedge \varphi_B(\bar{x}^{o}) \wedge Tr(\bar{x}, \bar{x}^{o}, \bar{x}')\right)$$

Abusing notation, we write $\mathscr{F}_{Tr}(\varphi_A)$ for $\mathscr{F}_{Tr}(\varphi_A, \varphi_A)$, and $\mathscr{F}(\varphi_A, \varphi_B)$ when *Tr* is clear from the context or is irrelevant. Using function $\mathscr{F}$, the CHC in (1) are equivalently expressed as two First Order Logic formulas:

$$\forall \bar{x}, \bar{x}', \bar{x}^{o} \cdot \mathscr{F}(Inv, Inv) \implies Inv(\bar{x}') \qquad\qquad \forall \bar{x} \cdot Inv(\bar{x}) \implies \neg Bad(\bar{x})$$

SPACER is a parameterized algorithm (or a set of rules) that is instantiated for a given logical theory T given three ingredients: (a) a model-producing satisfiability solver for T (i.e., an SMT solver that supports theory T), (b) an MBP procedure MBP for T, and (c) an interpolation procedure ITP for T. Here, we present a version that is limited to quantifier free solutions. Extension of SPACER for quantified solutions is described in [61].

The main data-structures operated by SPACER are a sequence of *may* summaries $[F_0, F_1, \ldots]$ called a *trace*, a *must* summary called R, and a queue of proof obligations Q. Each element F_i of a trace is called a frame, and each element $\ell \in F_i$ is called a lemma (or a may summary). Intuitively, F_i over-approximates all the states reachable by *Tr* in up to i steps (derivations). The set R, also called the *reachable states*, under-approximates all the reachable states. Finally, elements of Q, called proof-obligations, represent states the algorithm is trying to proof reachable or unreachable.

The rules defining SPACER are shown in Alg. 1. The rules are applied non-deterministically although, only some order of application guarantees progress. Each rule is presented as a guarded command "[*grd*] *cmd*", where *cmd* can be executed only if *grd* holds. If multiple guards are true, any one of the corresponding commands can be executed.

As described above, SPACER maintains a set of reachability queries Q, a sequence of may summaries $\{F_i\}_{i \in \mathbb{N}}$, and a must summary R. Intuitively, a query $\langle \varphi, i \rangle$ corresponds to checking if φ is reachable for recursion depth i, F_i over-approximates the reachable

Input: Formulas $Init(\bar{x}), Tr(\bar{x},\bar{x}^o,\bar{x}'), Bad(\bar{x})$

Output: An inductive invariant or UNSAFE

if $(Init \wedge Bad)$ *satisfiable* **then return** UNSAFE

// initialize data structures

$Q := \emptyset$ *// set of pairs* $\langle \varphi, i \rangle, i \in \mathbb{N}$

$N := 0$ *// max level, or recursion depth*

$F_0 = Init, F_i = \top, \forall i > 0$ *// may summary sequence*

$R = Init$ *// must summary*

forever *non-deterministically* **do**

(Candidate) [$(F_N \wedge Bad)$ *satisfiable* **]**
$Q := Q \cup \langle \varphi, N \rangle$, for some $\varphi \implies F_N \wedge Bad$

(MustPredecessor) [$\langle \varphi, i+1 \rangle \in Q$, $M \models \mathscr{F}(F_i, R) \wedge \varphi'$ **]**
$Q := Q \cup \langle \text{MBP}(\exists \bar{x}^o, \bar{x}' \cdot \mathscr{F}(F_i, R) \wedge \varphi', M), i \rangle$

(MayPredecessor) [$(\varphi, i+1) \in Q$, $M \models \mathscr{F}(F_i) \wedge \varphi'$ **]**
$Q := Q \cup \langle \text{MBP}(\exists \bar{x}, \bar{x}' \cdot \mathscr{F}(F_i) \wedge \varphi', M)[\bar{x}/\bar{x}^o], i \rangle$

(Leaf) [$(\varphi, i) \in Q$, $\mathscr{F}(F_{i-1}) \implies \neg\varphi'$, $i < N$ **]**
$Q := Q \cup \langle \varphi, i+1 \rangle$

(Successor) [$\langle \varphi, i+1 \rangle \in Q$, $M \models \mathscr{F}(R) \wedge \varphi'$ **]**
$R := R \vee \text{MBP}(\exists \bar{x}, \bar{x}^o \cdot \mathscr{F}(R) \wedge \varphi', M)[\bar{x}/\bar{x}']$

(NewLemma) [$\langle \varphi, i+1 \rangle \in Q$, $\mathscr{F}(F_i) \implies \neg\varphi'$ **]**
$F_j := F_j \wedge \text{ITP}(\mathscr{F}(F_i), \varphi')[\bar{x}/\bar{x}'], \forall j \leq i+1$

(Induction) [$(\varphi \vee \psi) \in F_i$, $\mathscr{F}(\varphi \wedge F_i) \implies \varphi'$ **]**
$F_j := F_j \wedge \varphi, \forall j \leq i+1$

(Unfold) [$F_N \implies \neg Bad$ **]** $N := N+1$

(Safe) [$F_{i+1} \implies F_i$ **]** **return** F_i

(Unsafe) [$(R \wedge Bad)$ *satisfiable* **]** **return** UNSAFE

Algorithm 1: Rule-based description of SPACER.

states for recursion depth i, and R under-approximates the reachable states. N denotes the current bound on recursion depth. The sequence of may summaries and N correspond to the *trace of approximations* and the maximum *level* in IC3/PDR, respectively. For convenience, let F_{-1} be $\bot$. $\text{MBP}(\varphi, M)$, for a formula $\varphi = \exists \bar{v} \cdot \varphi_{qf}$ and model $M \models \varphi_{qf}$, denotes the result of some MBP function associated with φ for the model M.

Alg. 1 initializes N to 0 and, F_0 and R to $Init$. **Candidate** initiates a backward search for a counterexample beginning with a set of states in Bad. The potential counterexample is expanded using either **MustPredecessor** or **MayPredecessor**. **MustPredecessor** *jumps over* the call $Inv(\bar{x}^o)$, in the last CHC of (1), utilizing the must summary R. **MayPredecessor**, on the other hand, creates a query for the call using the may summary of its calling context. **Leaf** moves an unreachable query to a higher recursion depth. **Suc-**

cessor updates R when a query is known to be reachable. **NewLemma** updates may summaries when a query is known to be unreachable. **Induction** strengthens may summaries using induction relative to F_i. **Unfold** increments the bound on the recursion depth. **Safe** returns F_i as invariant when the sequence of may summaries converges. **Unsafe** applies when the must summary intersects with *Bad*.

SPACER is sound and if MBP utilizes finite MBP functions, SPACER also terminates for a fixed N. Soundness follows from the fact that the following invariants are maintained by the main loop:

$$\begin{array}{rr} \mathit{Init} \implies F_0 & \forall 0 < i \leq N \cdot \mathscr{F}(F_{i-1}) \implies F_i \\ R \implies \mathscr{F}^N(\mathit{Init}) & \forall 0 < i \leq N \cdot F_{i-1} \implies F_i \\ \multicolumn{2}{c}{\forall 0 < i \leq N \cdot \mathscr{F}^i(\mathit{Init}) \implies F_i} \end{array}$$

Thus, $\{F_i\}_{i \in \mathbb{N}}$ and R, respectively, over- and under-approximate reachable states.

The rules in Alg. 1 leave out many important implementation details. For efficiency, queries are restricted to cubes (i.e., conjunction of literals). For Linear Arithmetic, the implementation relies on the fact that MBP is linear in time and space. Q is maintained as a priority queue, processing queries of smaller recursion depths first. Additional constraints are imposed on the rules and their ordering to ensure termination for a fixed N. For the rule **Unsafe**, the implementation also produces a counterexample in addition to returning UNSAFE.

4.2. Abstract Interpretation with CRAB

In this section, we first introduce CRAB [31] (CoRnucopia of ABstractions), a language-agnostic static analyzer based on the theory of Abstract Interpretation [25]. CRAB does not analyze directly LLVM bitcode but instead it analyzes a goto-based Control-Flow Graph language. This allows decoupling the analyzer from the input language so that it can be reused for analyzing other languages beyond LLVM bitcode (e.g., in [48]). Then, we describe CLAM (CRAB for Llvm Abstraction Manager), the static analyzer of LLVM bitcode based on CRAB, which is integrated in SEAHORN as one of its back-end solvers.

Note that we have decided to implement CRAB on top of a imperative language and not directly on Constrained Horn Clauses. This is motivated by the necessity to orient the analysis in Abstract Interpretation. That is, the interpreter needs to know the order in which to execute the instructions. While it is possible to map logical definitions into instructions, in practice, we chose to avoid this complication by going directly from LLVM to the intermediate representation used by CRAB.

4.2.1. CRAB *Target Language*

CRAB programs are written in the goto-based language described in Figure 5. A program P consists of a non-empty sequence of basic blocks, each one denoted by a unique identifier *bb*, containing zero or more instructions I in three-address form. Operands can only be one of these three basic types: integers, booleans, and pointers, or arrays of a basic type. All instructions are strongly typed. The language does not support floating point operations.

$$
\begin{array}{lll}
P & ::= & B^{+} \\
B & ::= & bb : I^{*}\ \textbf{goto}\ bb_1, \ldots bb_n \mid bb : I^{*}\ [\ \textbf{return}\ v_1, \ldots, v_n\ ] \\
I & ::= & I_a \mid I_b \mid I_p \mid I_A \mid v'_1, \ldots, v'_m := fun(v_1, \ldots, v_n) \\
 & & v := \textbf{havoc}\ () \mid \textbf{assume}\ (v_b) \mid \textbf{assume}\ (b) \mid \textbf{assert}\ (v_b) \mid \textbf{assert}\ (b) \\
I_a & ::= & v_i := a \\
 & & v_i := \textbf{sign\_extension}\ (v_i) \mid v_i := \textbf{zero\_extension}\ (v_i) \mid v_i := \textbf{truncate}\ (v_i) \\
 & & v_i := \textbf{booltoint}\ (v_b) \\
I_b & ::= & v_b := b \mid v_b := \textbf{inttobool}\ (v_i) \\
I_p & ::= & v_p := p \mid v_p := \textbf{alloc}\ (sz) \mid v_s := \textbf{load}\ (v_p) \mid \textbf{store}\ (v_s, v_p) \mid v_p := \&\mathrm{fun} \\
I_A & ::= & v_A := \textbf{array\_init}\ (v_i, v'_i, v_s, sz, endian) \mid v_s := \textbf{array\_select}\ (v_A, v_i, sz, endian) \\
 & & \textbf{array\_write}\ (v_A, v_i, v_s, sz, endian) \mid v'_A := v_A \\
a & ::= & n \mid v_i \mid a_1\ op_a\ a_n \\
b & ::= & \mathsf{true} \mid \mathsf{false} \mid \neg\ b \mid b_1\ op_b\ b_2 \mid a_1\ op_r\ a_2 \mid p_1\ op_p\ p_2 \\
p & ::= & \mathsf{null} \mid v_p + a
\end{array}
$$

Figure 5. CRAB goto-based language to represent Control Flow Graphs.

Integer, boolean, pointer, and array variables are denoted with symbols v_i, v_b, v_p, v_A, respectively. Variables of any type are denoted by v. Scalar (non-array) variables are denoted by v_s. Integer variables are sized (i.e., of different bit-width). The set of integer, boolean, pointer, and array variables are disjoint.

Arithmetic and boolean instructions. Arithmetic and boolean expressions are described by a and b. CRAB supports standard operations op_a and op_b for these expressions. For arithmetic expressions, CRAB supports addition, subtraction, multiplication, signed/unsigned division, signed/unsigned remainder, and standard bitwise operations: `and`, `or`, `xor`, left shift, logical and arithmetic right shift. For boolean expressions, CRAB supports the operations `and`, `or`, and `xor`.

Control flow and assertions. Control flow is modeled by **goto** and **assume** instructions. The instruction $v := \textbf{havoc}()$ assigns non-deterministically any value allowed by v's type to v. Properties can only be defined by adding **assert** instructions.

Pointer instructions. The instruction $v_p := \textbf{alloc}(sz)$ allocates a fresh memory object of size sz and returns a pointer to it. The instructions $v_s := \textbf{load}(v_p)$ and $\textbf{store}(v_s, v_p)$ read and write memory. Pointer arithmetic can be expressed by $v_p := v'_p + a$. CRAB also supports function pointers $v_p :=$ &fun. For pointer comparisons op_p, CRAB supports pointer equality and disequality.

Array instructions. CRAB language supports unidimensional arrays. The importance of arrays is inherited from the importance of arrays in imperative languages and even more important, because the program memory can be modeled as an array. Arrays are interpreted as sequences of consecutive bytes which are disjoint from each other. We describe informally the semantics of the array operations. We define first $\mathsf{BS}(v_A, v_i, sz, endian)$ as the byte sequence:

$$
\begin{cases}
v_A[v_i] \cdot v_A[v_i+1] \cdots v_A[v_i+sz-1] & \text{if } endian = \mathsf{big} \\
v_A[v_i+sz-1] \cdot v_A[v_i+sz-2] \cdots v_A[v_i] & \text{if } endian = \mathsf{little}
\end{cases}
$$

Similarly, we define $\mathsf{BS}(v_s, endian)$ as the byte sequence:

$$\begin{cases} v_s(0)\cdots v_s(n-1) & \text{if } endian = \mathsf{big} \\ v_s(n-1)\cdots v_s(0) & \text{if } endian = \mathsf{little} \end{cases}$$

The instruction v_A := **array_init** $(v_i, v_i', v_s, sz, endian)$ creates a fresh array v_A such that for all $v_i \leq j < v_i' \wedge (j \bmod sz = 0)$, each byte sequence $\mathsf{BS}(v_A, j, sz, endian)$ is equal to $\mathsf{BS}(v_s, endian)$. Array reads v_s := **array_select** $(v_A, v_i, sz, endian)$ assigns the byte sequence $\mathsf{BS}(v_A, v_i, sz, endian)$ to $\mathsf{BS}(v_s, endian)$. **array_write** $(v_A, v_i, v_s, sz, endian)$ writes the byte sequence $\mathsf{BS}(v_s, endian)$ into $\mathsf{BS}(v_A, v_i, sz, endian)$. Finally, v_A' := v_A assigns all contents of v_A to v_A'.

Endianess can be optionally provided. However, if it is not available then array abstract domains are limited when reasoning about byte aliasing because they cannot make any assumption about endianess.

Function calls. CRAB assumes call-by-value parameter passing. Functions can return multiple values. This is specially useful for purifying functions. *Function purification* converts functions into new equivalent functions that have no side effects.

Conversion between types. Conversion between operand types is allowed but it must be done through explicit casts. CRAB supports sign and zero extension, truncation, and conversions between boolean and integers. Conversion between integers and pointers is not currently supported.

CRAB *language design choices.* The design of the CRAB language has been carefully chosen based on our experience in building abstract interpreters and front-ends. For instance, the language distinguishes between boolean and integer variables although boolean can be also modeled as integers if desired. The distinction between boolean and integers can make easier the translation to the CRAB language if the front-end already makes that separation. Moreover, it can simplify the code of an abstract interpreter because boolean and arithmetic instructions can be analyzed by different abstract domains: boolean instructions with a finite domain and numerical instructions with a numerical abstract domain. The distinction between pointer and arrays instructions is another good example. Typically, abstract domains reasoning about pointers (e.g., [85, 100]) are very different from domains reasoning about arrays (e.g., [29, 49, 65]). The former focuses on aliasing while the latter focuses more on the problem of weak versus strong updates. Again, having specialized instructions for pointers and arrays can simplify the code of the abstract interpreter. Moreover, there are situations where either the input language is simple enough that aliasing is not an issue (e.g., [48]) or the front-end can resolve aliasing at translation time (see Section 4.2.7). For those cases, array domains are sufficient to reason precisely about memory.

Compared to other intermediate representations such as LLVM IR, control flow is expressed in a more declarative way by having **goto** and **assume** instead of conditional branches. Unlike LLVM IR, CRAB language is not intended to be executed, and thus, it allows expressing non-determinism through **havoc** instructions which is very useful for program abstractions (e.g., model a cast from an integer to a pointer). Another key difference with LLVM IR is that the CRAB language does not require the input program to be written in Static Single Assignment (SSA), and, therefore, it does not have ϕ-nodes. This special instruction is used to represent all the possible values of a variable can take at a merge point in the Control Flow Graph (CFG). The analysis of ϕ-nodes using abstract

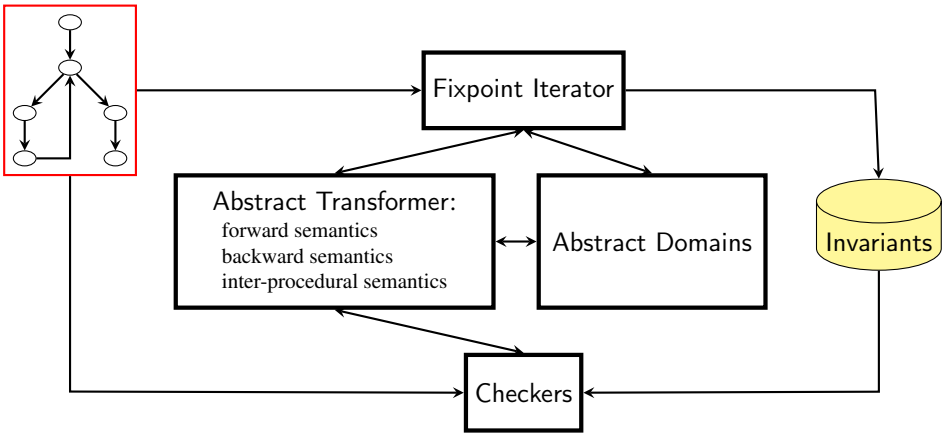


Figure 6. CRAB architecture.

interpretation is specially challenging with relational numerical domains [43]. For that reason, Crab language does not allow ϕ-nodes.

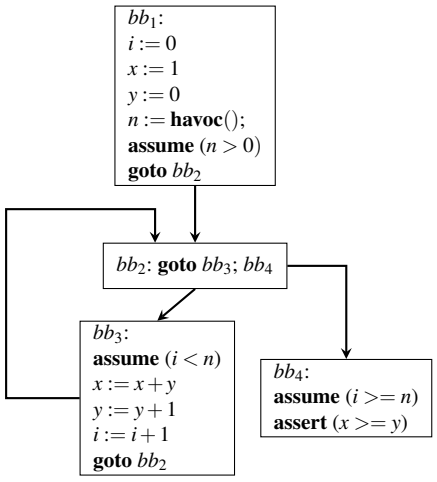
4.2.2. Tool Architecture

The design of Crab is very similar to other abstract interpreters such as ASTRÉE [14], CLOUSOT [38] and IKOS [16]. Crab is parametric both in the fixpoint iterators and abstract domains. The main architecture of the tool is depicted in Figure 6. We omit for now the details about how to adapt this architecture to inter-procedural analysis. This is described in in Section 4.2.6. Crab has two operation modes: the *inference mode* and the *checking mode*.

Inference mode. CRAB takes as input the CFG as described in previous section. Then, it solves iteratively the semantic equations extracted directly from the CFG. Solving these equations is performed by the Fixpoint Iterator. The fixpoint iterator is in charge of finding good iteration strategies and in charge of applying widening and narrowing in effective ways, while optimizing time and memory consumption. Solving semantic equations requires to both interact with Abstract Domains (e.g., join, meet, widening, and narrowing) and with Abstract Transformer to apply the corresponding semantics to each CFG instruction (e.g., forward semantics, backward semantics, or inter-procedural semantics). Figure 7 shows an example of semantic equations extracted from a sample CFG.

Finally, after Fixpoint Iterator has found a stable solution (a.k.a. invariants), these are stored in an *invariant database*. When inference mode is enabled, no warning is displayed when some possible error is detected. This is because either the fixpoint has not been reached yet and it might be unsound to report the program is safe, or a warning can be ruled out after refinement using narrowing or dual narrowing operators [23, 28].

Checking mode. Upon completion of the inference phase, CRAB uses the invariants stored in the database to check if certain properties can be violated, issuing warning messages. This is done by Checkers. Currently, CRAB can perform some built-in checks (division by zero, null-dereference, etc) and user-defined assertions (**assert** instructions).



$$X_0 = \top$$
$$X_1 = \mathsf{forget}(n, \mathsf{post}(\{i = 0; x := 1; y := 0\}, X_0)) \sqcap (n > 0)$$
$$X_2 = X_2 \bigtriangledown (X_1 \sqcup X_3)$$
$$X_3 = \mathsf{post}(\{x := x + y; y := y + 1; i := i + 1\}, (X_2 \sqcap (i < n)))$$
$$X_4 = X_2 \sqcap (i \geq n) \sqcap (x < y)$$

Figure 7. Crab CFG and its forward semantic equations. Each X_i $(1 \leq i \leq 4)$ represents the invariants that hold at the exit of block bb_i. The initial abstract state is $X_0 = \top$ (top). The assertion holds if $X_4 = \bot$ (bottom).

For efficiency, invariants are only stored per CFG basic block. Thus, instructions might be re-analyzed but only up to the end of each basic block. This saves us from storing invariants per instruction at a very small cost.

4.2.3. Fixpoint Iterator

CRAB computes first a weak topological ordering (WTO) of the CFG following Bourdoncle's algorithm [15]. The WTO produces a good order in which basic blocks should be analyzed, and the set of basic blocks in the CFG where the fixpoint algorithm needs to apply widening to ensure termination.

The current fixpoint iterator used in Crab is based on [4], and it interleaves widening and narrowing operations for each inner loop until reaching convergence before analyzing outer loops. Thresholds are used to improve the precision of the widening (as in [14, 38]). The thresholds are collected statically from the constants appearing in **assume** and **assert** instructions.

The Fixpoint iterator is very generic since it focuses only on solving semantic equations. The particular semantics is given by the Abstract Transformer. The advantage is that it can be replaced by any method as long as one focuses on iterative solving techniques [50, 51, 64, 76, 83, 94].

4.2.4. Abstract Domains

Abstract domains are in charge of interpreting in the abstract the operators $\sqcup, \bigtriangledown, \ldots$, and transfer functions appearing during the solving of semantic equations. A very simplified view of the abstract domain interface in CRAB is shown in Figure 8.

The forward (post) and backward (pre) transfer functions are used by Abstract Transformer, and the Fixpoint iterator calls the other operations while computing the fixpoint of the semantic functions. The inter-procedural semantics is implemented by the Abstract Transformer calling directly abstract domain operations (project, $\sqcap$, forget, ...).

Constructors
makeTop : D
makeBottom : D
Forward and backward transfer functions
post : $Instr^+ \times D \mapsto D$
pre : $Instr^+ \times D \mapsto D$
forget : $Var^+ \times D \mapsto D$
project : $Var^+ \times D \mapsto D$
Fixpoint iterator operations
isBottom : $D \mapsto \mathbb{B}$ (emptiness test)
$\sqsubseteq : D \times D \mapsto \mathbb{B}$ (inclusion test)
$\sqcup : D \times D \mapsto D$ (join)
$\sqcap : D \times D \mapsto D$ (meet)
$\bigtriangledown : D \times D \mapsto D$ (widening)
$\bigtriangleup : D \times D \mapsto D$ (narrowing)

Figure 8. Simplified Abstract Domain API.

Intervals and Congruences. *Intervals* [24] expresses constraints of the form $x = [lb, ub]$ meaning that $lb \leq x$ and $x \leq ub$ where x is an integer variable and $lb, ub \in \mathbb{Z} \cup \{-\infty, +\infty\}$.

Congruences [52] expresses constraints of the form $a\mathbb{Z} + b$ where a and b are integers. For instance, all even (odd) numbers are represented as $2\mathbb{Z} + 0$ ($2\mathbb{Z} + 1$). Both domains are combined via a reduced product as described in [52].

These *non-relational* domains are represented by environments from variables to abstract values. CRAB uses *functional maps* [89] to implement efficiently these environments as in [14, 16, 38].

Zones (a.k.a Difference-Bounds Matrices) [84] expresses constraints of the form $x - y \leq k$, where x, y are integer variables and $k \in \mathbb{Z}$. CRAB uses an efficient sparse implementation of difference-bounds matrices that achieves sparsity by dynamically separating interval constraints from constraints that can only be expressed through differences [45]. In our experience, Zones is one of our most important domains because it can compute non-trivial relationships between variables while still being efficient in practice (see e.g., [48]).

Flat Boolean Domain is a finite lattice $\bot \leq \texttt{T} \leq \top$, $\bot \leq \texttt{F} \leq \top$ that discovers which boolean variables are definitely true `T` or false `F`. This domain is always combined with a numerical domain so that information can flow between integer to boolean variables.

DisInt [38] is an extension of Intervals to a finite disjunction. Elements in this domain are normalized sequences of non-overlapping, sorted intervals $[a_o, b_0], \ldots, [a_n, b_n]$ such that only a_0 can be $-\infty$ and b_n can be $+\infty$. This domain retains the scalability of the Interval domain while being able to reason about simple disequalities. For instance, the disequality $\neq 0$ can be expressed by the sequence of intervals $[-\infty, -1]$ and $[1, +\infty]$.

Boxes [55] expresses finite boolean combinations of Intervals. Boxes provides an efficient implementation of the exact disjunction of Intervals based on Linear Decision Diagrams [17]. This domain is very useful for path-sensitive analyses. This domain reasons

simultaneously about both boolean and integer variables producing much more precise results than the Flat Boolean Domain combined with, for instance, Intervals.

Numerical domain with uninterpreted functions. The *Terms* domain [44] can improve the precision of a numerical domain by inferring equivalence amongst sub-expressions based on the theory of uninterpreted functions. Terms is strictly more precise than [87], and it can enhance numerical domains in different ways by: (a) providing some relational information (equalities) to domains such as Intervals and Congruences, (b) improving precision in presence of non-linear operations (e.g., $x \leq 10 \wedge y = \sqrt{x} + y^2 \wedge z = \sqrt{x} + y^2 \rightarrow x \leq 10 \wedge y = z$), and (c) improving precision in presence of array operations (e.g., $b = write(a,i,x,sz) \wedge y = select(a,i,sz) \rightarrow x = y$).

Intervals over machine arithmetic. Most CRAB numerical domains reason about unbounded integers. This forces us to check for integer overflow in order to produce sound verification results. The exception is the Wrapped Interval Domain [88] which infers interval constraints obeying the laws of machine arithmetic, and thus, it can produce correct intervals in the presence of integer wraparounds.

Interface to Apron and Elina domains. CRAB provides interfaces to external abstract domains libraries such as Apron [69] and Elina [95]. Thus, domains such as Octagons [86] ($\pm x \pm y \leq k$, where x, y are integer variables and $k \in \mathbb{Z}$) and Polyhedra [30] (linear inequalities of the form $\sum_i c_i \cdot x_i \leq k_i$ where x_i are integer variables, $c_i, k_i \in \mathbb{Z}$) are also available.

Nullity domain is a finite lattice $\bot \leq \texttt{N} \leq \top$, $\bot \leq \texttt{NN} \leq \top$ that discovers which pointer variables are definitely null `N` or non-null `NN`.

Array content domains lift abstract domains for scalar variables to reason about arrays. CRAB provides several array content domains that strike different balances between precision and cost. *Array Smashing* [14] treats the whole array as a single symbolic variable. The transfer function for **array_write** can only weaken the previous abstract state (*weak update*). Array Smashing can represent universally quantified invariants if they hold uniformly for all array elements.

On the other extreme, *Array Expansion* [14] treats individually each array element as a single scalar variable. This domain can be more precise because the transfer function for **array_write** can overwrite the old value (*strong update*) and can reason about byte aliasing. However, it might not scale if arrays are too large, and it cannot express universally quantified properties.

Similar to ASTRÉE, CRAB also implements the combination of both domains via a reduced product. Arrays are initially populated using strong updates by the Array Expansion domain. If the size of the array is greater than certain threshold or arrays are accessed using non-constant indexes then they are smashed. A smashed array can be expanded again as in [14]. In CRAB, once an array is smashed it is not expanded again.

CRAB also provides an array content domain [41] based on *Array Partitioning* [49, 65]. The domain is useful when an invariant does not hold for all array elements but instead on some contiguous segment. The domain selects a small set of partition variables, maintaining disjunctive information about properties which hold over the segments delimited by the partition variables. This domain can express properties such as array sortedness but it can suffer from scalability issues. More efficient array domains such as [29] can be also implemented in CRAB.

Combination of domains. Most of the abstract domains described above are typically not enough to prove non-trivial properties. However, their combination can produce very powerful analyses [26]. The main method for combining numerical abstract domains in CRAB is the *reduced product* [25]. Given two domains D_1 and D_2, the reduced product is equivalent to running simultaneously both domains while communicating information between the two domains. This communication is called *reduction*. CRAB provides a generic reduced product domain that redirects each abstract operation to each sub-domain, performing a simple reduction: $\bot$ if any of the sub-domains is $\bot$. More complex reductions are carefully implemented on a case-by-case basis (e.g., Intervals and Congruences, Terms and Zones, Array Smashing and Expansion, etc).

4.2.5. Backward Analysis

When an invariant is too weak to prove an assertion, we can propagate backwards the predecessors of the error states and use the abstract states at those points to prove that there is no an execution starting from the entry of the program that can reach an error state. This approach is good at handling some disjunctive invariants which many of CRAB abstract domains cannot represent precisely. A typical scenario is when an assertion after a join point might not be provable due to loss of precision at the join.

The CRAB backward analysis is based on computing necessary preconditions. A *necessary precondition* of a set of states F is the set of initial states that guarantee that some of its executions will stay in F. Similar to [92], CRAB computes in the abstract necessary preconditions starting from the set of error states. These error states are obtained by representing in the abstract the negation of an assertion condition. If the set of initial states is empty then the set of error states must be unreachable, and thus, the assertion definitely holds.

As described in [27], the precision of the backward analysis can be improved by considering only preconditions that might be reachable from the entry of the program. For any basic block b, CRAB intersects in the abstract ($\sqcap$ operator) the preconditions from the error states at b with the invariants that hold at b. Next, a new forward analysis is run starting from the new preconditions computed by the backward analysis in an attempt to produce more precise invariants. This interleaving process between a forward and backward analysis gives an infinite descending chain of approximated preconditions and invariants whose termination is ensured by narrowing.

4.2.6. Inter-Procedural Analysis

The architecture shown in Figure 6 is limited to intra-procedural analysis. However, inter-procedural analysis is also available in CRAB. Based on our experience, different programs are more amenable to different inter-procedural analyses. To support this view, CRAB is also parametric on the inter-procedural analysis. For a new inter-procedural analysis only these two main steps need to be implemented in CRAB:

- Define the inter-procedural semantics for **call** and **return**.
- Find an effective ordering to traverse the call graph while computing globally stable solutions. Depending on the program, these are the common cases that might need to be considered:
  (1) Incomplete call graph and recursive functions
  (2) Incomplete call graph and non-recursive functions

(3) Complete call graph and recursive functions
(4) Complete call graph and non-recursive functions

Cases (1) and (2) require running simultaneously a pointer analysis together with the abstract domain chosen to reason about the desired property in order to resolve indirect calls. Case (3) can be solved by computing a global fixpoint for each strongly connected component in the call graph. Weak topological ordering [15] can be used to identify widening points. Finally, (4) is the simplest case because the call graph is a directed acyclic graph (DAG).

CRAB implements an inter-procedural analysis based on CGS [101]. The inter-procedural analysis makes the main assumption that the call graph is complete and thus, all the function calls have been already resolved by the client (Section 4.2.7 described how we can ensure that for LLVM programs). A cycle in the call graph is treated by analyzing all the functions in the cycle in an intra-procedural manner. Therefore, the call graph analyzed by CRAB can be considered in practical terms as a DAG. The analysis performs a *context-insensitive*, *summary-based* inter-procedural analysis consisting of two phases:

- Bottom-up (callees before callers): traverse the call graph in reverse topological order while computing summaries for each function. *Summaries* are abstract states relating input with output function parameters. Summaries are computed by first inferring invariants for the function using the intra-procedural analysis and then by producing the actual summary during the transfer function of **return**. While computing invariants, the analysis might need to reuse other summaries from callees. This is implemented in the transfer function of **call**.
- Top-down (callers before callees): traverse the call graph in topological order starting from main. At each call, the inter-procedural semantics for **call** reuses the summary of the callee after formal/actual parameters renaming, as done already during the bottom-up traversal. Moreover, it stores the preconditions associated to that call in the callee. Note that at the time a function is analyzed, all its callers have been already analyzed, and thus, all the preconditions are available. CRAB is context-insensitive because it joins all the preconditions.

The inter-procedural analysis is quite fast since each function is analyzed exactly once. However, the analysis can be imprecise since it is context-insensitive. A context-sensitive analysis can be implemented by keeping separate the preconditions of each function call and then running different analyses starting from each precondition.

4.2.7. Clam

CLAM (CRAB for Llvm Abstraction Manager) is an abstract interpreter for LLVM based on CRAB. The main tasks performed by CLAM are:

1. Translating each LLVM function to a CRAB goto-based Control-Flow Graph.
2. Running CRAB analyses.
3. Assertion checking and/or communicating CRAB invariants to other SeaHorn back-end solvers.

CLAM allows users to choose among several parameters such as the abstract domain, fixpoint parameters (widening delay, number of thresholds, etc.), and whether backward or inter-procedural analysis should be enabled or not. SEAHORN users can choose CLAM as the only back-end engine to discharge proof obligations. However, even if the abstract domain can express precisely the program semantics, due to the join and widening operations, it might lose some precision during the verification. As a consequence, CLAM alone might not be sufficient as a back-end engine. Instead, a more suitable job for CLAM is to supply program invariants to the other engines (e.g. SPACER). For this, the integration between CLAM and SPACER has been carefully tuned. CLAM invariants can be used by SPACER in two different ways. First, invariants can be added as permanent lemmas (i.e., initial may summaries) to initialize each frame F_i. In this case, the exploration done by SPACER is limited to states that satisfy the invariants discovered by CLAM. Second, CLAM invariants can be added only to restrict the transition relation during the **Induction** rule. This mode does not interfere with exploration, but can improve generalization done by the rule.

The translation of integer instructions is straightforward. Most of LLVM instructions with integer operands have their CRAB counterparts with the exceptions of ϕ and branch instructions which are replaced with CRAB assignments and **assume**, respectively. Similarly, CLAM can translate directly LLVM pointer instructions to CRAB pointer instructions (**alloc**, **load**, and **store**). Note that this syntactic-guided translation approach is pretty simple, but it relies entirely on CRAB to reason about both LLVM registers and memory.

Alternatively, CLAM can perform much of the memory reasoning at translation time by leveraging SEADSA, which is already used during the generation of Constrained Horn Clauses. CLAM can use SEADSA to *disambiguate memory*, i.e., resolve pointer aliasing, which allows us to:

1. Resolve all indirect calls, producing a complete call graph.
2. Perform function *purification*, eliminating all function side-effects.
3. Translate LLVM `load` and `store` instructions to CRAB array instructions **array_select** and **array_write**, respectively.

Steps 1 and 2 greatly simplify CRAB inter-procedural analysis. Step 3 allows leveraging powerful CRAB array domains to infer rich invariants about integer memory values, without complex abstract domains that would need also to reason about pointer aliasing. In our experience, this relatively simple approach has been quite effective at reasoning about C programs.

5. Conclusions

We have presented SEAHORN, a software verification framework with a modular design that separates the concerns of the syntax of the language, its operational semantics, and the verification semantics. SEAHORN builds upon two verification engines: SPACER and CRAB. Both SPACER and CRAB represent the state-of-the-art in SMT-based Model Checking and Abstract Interpretation, respectively.

We believe that SEAHORN is a versatile and highly customizable framework that can help significantly in the time-consuming process of building new tools by allowing researchers experimenting only on their particular techniques of interest.

The flexibility and practicality of SEAHORN have been demonstrated by ourselves and other researchers over several projects. In our seminal work [58], we use SEAHORN to prove proper API usage of Linux device drivers and memory safety of autopilot code. In [99], SEAHORN is extended to go beyond safety properties for proving termination of programs. In [70], SEAHORN is used to find code inconsistencies, code fragments without normal terminating executions. In [71], SEAHORN is used to prove safety of smart contracts. In this case, our Clang-based front-end was replaced with an in-house developed front-end that translates Solidity smart contracts directly to LLVM bitcode. In our most recent work, we have used SEAHORN to prove equivalence of x86 executable programs [36]. We use McSema [98] to translate x86 code to LLVM bitcode and then use the SEAHORN Bounded Model Checking engine to prove equivalence of a program and an equivalent variant that is more resilient to cyber-security attacks.

While SEAHORN is already a full featured verification engine, significant work remains to improve both usability and scalability. From the usability perspective, the main questions are around user communication with the tool. Currently, each new property requires non-trivial encoding of a problem domain to a low-level language. Adding support for a new property is non-trivial research-driven effort. On the other side, the results from SEAHORN are difficult to interpret by an average developer. Recently, we have proposed that in the case of counterexamples, the tool must produce an executable that a developer can examine [47]. Note that this is quite different than producing failing inputs, since the executable contains not just the inputs to the original program, but also an executable model of the whole verification environment. While this is a good first step, more work remains to make this practical and robust in the presence of complex programming features including procedures and dynamic memory allocation. From the scalability perspective, dealing with dynamic memory allocation and multi-threaded code are currently the weakest links. We hope to address both by a more modular (but possibly incomplete) reasoning techniques further combining Model Checking and Abstract Interpretation.

6. Acknowledgments

SEAHORN would have not been possible without numerous great collaborators. First of all, we would like to thank Temesghen Kashai for being one of the original developers of SEAHORN and main contributor for several years. We would like also to thank (alphabetical order) Nikolaj Bjørner, Graeme Gange, Jeff Gennari, Anvesh Komuravelli, Jakub Kuderski, Peter Schachte, Edward Schwartz, Harald Søndergaard, and Peter Stuckey.

References

[1] A. Albarghouthi, A. Gurfinkel, and M. Chechik. Craig Interpretation. In *SAS*, pages 300–316, 2012.

[2] A. Albarghouthi, A. Gurfinkel, and M. Chechik. Whale: An interpolation-based algorithm for inter-procedural verification. In *VMCAI*, pages 39–55, 2012.

[3] A. Albarghouthi, Y. Li, A. Gurfinkel, and M. Chechik. Ufo: A framework for abstraction- and interpolation-based software verification. In *CAV*, pages 672–678, 2012.

[4] G. Amato and F. Scozzari. Localizing widening and narrowing. In *SAS*, pages 25–42, 2013.

[5] S. Arlt, C. Rubio-González, P. Rümmer, M. Schäf, and N. Shankar. The gradual verifier. In *NFM*, pages 313–327, 2014.

[6] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani. Software model checking via large-block encoding. In *FMCAD*, pages 25–32, 2009.

[7] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker blast. *STTT*, 9(5-6):505–525, 2007.

[8] D. Beyer and M. E. Keremoglu. Cpachecker: A tool for configurable software verification. In *CAV*, pages 184–190, 2011.

[9] N. Bjørner and A. Gurfinkel. Property directed polyhedral abstraction. In *VMCAI*, pages 263–281, 2015.

[10] N. Bjørner, A. Gurfinkel, K. L. McMillan, and A. Rybalchenko. Horn clause solvers for program verification. In *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*, pages 24–51, 2015.

[11] N. Bjørner and M. Janota. Playing with quantified satisfaction. In *LPAR*, pages 15–27, 2015.

[12] N. Bjørner, K. L. McMillan, and A. Rybalchenko. Program verification as satisfiability modulo theories. In *SMT*, pages 3–11, 2012.

[13] N. Bjørner, K. L. McMillan, and A. Rybalchenko. On solving universally quantified horn clauses. In *SAS*, pages 105–125, 2013.

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

[15] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *In Proceedings of the International Conference on Formal Methods in Programming and their Applications*, pages 128–141. Springer-Verlag, 1993.

[16] G. Brat, J. A. Navas, N. Shi, and A. Venet. IKOS: A framework for static analysis based on abstract interpretation. In *SEFM*, pages 271–277, 2014.

[17] S. Chaki, A. Gurfinkel, and O. Strichman. Decision diagrams for linear arithmetic. In *FMCAD*, pages 53–60, 2009.

[18] S. Chatterjee, S. K. Lahiri, S. Qadeer, and Z. Rakamaric. A reachability predicate for analyzing low-level software. In *TACAS*, pages 19–33, 2007.

[19] A. Cimatti, A. Griggio, and R. Sebastiani. Efficient generation of craig interpolants in satisfiability modulo theories. *ACM Trans. Comput. Log.*, 12(1):7:1–7:54, 2010.

[20] E. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs. In *TACAS*, pages 168–176, 2004.

[21] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. Vcc: A practical system for verifying concurrent c. In *TPHOL*, pages 23–42, 2009.

[22] L. Cordeiro, B. Fischer, and J. Marques-Silva. Smt-based bounded model checking for embedded ANSI-C software. *IEEE Trans. Software Eng.*, 38(4):957–974, 2012.

[23] P. Cousot. Abstracting induction by extrapolation and interpolation. In *VMCAI*, pages 19–42, 2015.

[24] P. Cousot and R. Cousot. Static Determination of Dynamic Properties of Programs. In *Proceedings of the second international symposium on Programming, Paris, France*, pages 106–130, 1976.

[25] 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.

[26] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282, 1979.

[27] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *J. Log. Program.*, 13(2&3):103–179, 1992.

[28] P. Cousot and R. Cousot. Comparing the galois connection and widening/narrowing approaches to abstract interpretation. In *PLILP*, pages 269–295, 1992.

[29] P. Cousot, R. Cousot, and F. Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *POPL*, pages 105–118. ACM, 2011.

[30] P. Cousot and N. Halbwachs. Automatic discovery of linear constraints among variables of a program. In *POPL*, pages 84–97. ACM, 1978.

[31] Crab: A language-agnostic library for Abstract Interpretation. Available from `https://github.com/seahorn/crab`.

[32] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-c: A software

analysis perspective. In *SEFM*, pages 233–247, 2012.

[33] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. VeriMAP: A tool for verifying programs through transformations. In *TACAS*, pages 568–574, 2014.

[34] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.

[35] R. DeLine and K. R. M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.

[36] B. Dutertre, I. Mason, and J. A. Navas. Proving equivalence of x86 programs with McSema and SeaHorn, 2018. Blog available at `http://seahorn.github.io/seahorn/mcsema/equivalence/x86/binary/2018/12/12/seahorn-and-mcsema.1.html`.

[37] Ethereum. The Ethereum EVM JIT. Available at `https://github.com/ethereum/evmjit`.

[38] M. Fähndrich and F. Logozzo. Static contract checking with abstract interpretation. In *FoVeOOS*, pages 10–30, 2010.

[39] G. Fedyukovich, A. Gurfinkel, and A. Gupta. Lazy but effective functional synthesis. In *VMCAI*, pages 92–113, 2019.

[40] G. Fedyukovich, A. Gurfinkel, and N. Sharygina. Property directed equivalence via abstract simulation. In *CAV*, pages 433–453, 2016.

[41] G. Gange, J. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey. A partial-order approach to array content analysis. Technical report, `https://arxiv.org/pdf/1408.1754.pdf`, 2014.

[42] G. Gange, J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey. Failure tabled constraint logic programming by interpolation. *TPLP*, 13(4-5):593–607, 2013.

[43] G. Gange, J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey. Horn clauses as an intermediate representation for program analysis and transformation. *TPLP*, 15(4-5):526–542, 2015.

[44] G. Gange, J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey. An abstract domain of uninterpreted functions. In B. Jobstmann and K. R. M. Leino, editors, *VMCAI*, pages 85–103, 2016.

[45] G. Gange, J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey. Exploiting sparsity in difference-bound matrices. In *SAS*, pages 189–211, 2016.

[46] P. Garoche, T. Kahsai, and C. Tinelli. Incremental invariant generation using logic-based automatic abstract transformers. In *NASA NFM*, pages 139–154, 2013.

[47] J. Gennari, A. Gurfinkel, T. Kahsai, J. A. Navas, and E. J. Schwartz. Executable counterexamples in software model checking. In *VSTTE*, pages 17–37, 2018.

[48] E. Gershuni, N. Amit, A. Gurfinkel, N. Narodytska, J. A. Navas, N. Rinetzky, L. Ryzhyk, and M. Sagiv. Simple and precise static analysis of untrusted linux kernel extensions. In *PLDI*, 2019.

[49] D. Gopan, T. Reps, and M. Sagiv. A framework for numeric analysis of array operations. In *POPL*, pages 338–350. ACM, 2005.

[50] D. Gopan and T. W. Reps. Lookahead widening. In *CAV*, pages 452–466, 2006.

[51] D. Gopan and T. W. Reps. Guided static analysis. In *SAS*, pages 349–365, 2007.

[52] P. Granger. Static analysis of arithmetical congruences. *International Journal of Computer Mathematics*, 30:165–190, 1989.

[53] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, pages 405–416, 2012.

[54] A. Griggio, T. T. H. Le, and R. Sebastiani. Efficient interpolant generation in satisfiability modulo linear integer arithmetic. *Logical Methods in Computer Science*, 8(3), 2010.

[55] A. Gurfinkel and S. Chaki. Boxes: A symbolic abstract domain of boxes. In *SAS*, pages 287–303, 2010.

[56] A. Gurfinkel and S. Chaki. Combining predicate and numeric abstraction for software model checking. *STTT*, 12(6):409–427, 2010.

[57] A. Gurfinkel, S. Chaki, and S. Sapra. Efficient Predicate Abstraction of Program Summaries. In *NFM*, pages 131–145, 2011.

[58] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas. The SeaHorn Verification Framework. In *CAV*, pages 343–361, 2015.

[59] A. Gurfinkel, T. Kahsai, and J. A. Navas. SeaHorn: A framework for verifying C programs - (competition contribution). In *TACAS*, 2015.

[60] A. Gurfinkel and J. A. Navas. A context-sensitive memory model for verification of C/C++ programs. In *SAS*, pages 148–168, 2017.

[61] A. Gurfinkel, S. Shoham, and Y. Vizel. Quantifiers on demand. In *ATVA*, pages 248–266, 2018.

[62] A. Gurfinkel and Y. Vizel. DRUPing for interpolates. In *FMCAD*, pages 99–106, 2014.

[63] A. Gurfinkel, O. Wei, and M. Chechik. Model checking recursive programs with exact predicate abstraction. In *ATVA*, pages 95–110, 2008.

[64] N. Halbwachs and J. Henry. When the decreasing sequence fails. In *SAS*, pages 198–213, 2012.

[65] N. Halbwachs and M. Péron. Discovering properties about arrays in simple programs. In *PLDI*, pages 339–348. ACM, 2008.

[66] M. Heizmann, J. Christ, D. Dietsch, E. Ermis, J. Hoenicke, M. Lindenmann, A. Nutz, C. Schilling, and A. Podelski. Ultimate Automizer with SMTInterpol - (Competition Contribution). In *TACAS*, pages 641–643, 2013.

[67] K. Hoder and N. Bjørner. Generalized property directed reachability. In *SAT*, pages 157–171, 2012.

[68] J. Jaffar, V. Murali, J. A. Navas, and A. E. Santosa. TRACER: A symbolic execution tool for verification. In *CAV*, pages 758–766, 2012.

[69] B. Jeannet and A. Miné. A library of numerical abstract domains for static analysis. In A. Bouajjani and O. Maler, editors, *CAV*, volume 5643 of *LNCS*, pages 661–667. Springer, 2009.

[70] T. Kahsai, J. A. Navas, D. Jovanovic, and M. Schäf. Finding inconsistencies in programs with loops. In *LPAR*, pages 499–514, 2015.

[71] S. Kalra, S. Goel, M. Dhawan, and S. Sharma. ZEUS: analyzing safety of smart contracts. In *NDSS*, 2018.

[72] A. Katis, G. Fedyukovich, H. Guo, A. Gacek, J. Backes, A. Gurfinkel, and M. W. Whalen. Validity-guided synthesis of reactive systems from assume-guarantee contracts. In *TACAS*, pages 176–193, 2018.

[73] A. Komuravelli, N. Bjørner, A. Gurfinkel, and K. L. McMillan. Compositional verification of procedural programs using horn clauses over integers and arrays. In *FMCAD*, pages 89–96, 2015.

[74] A. Komuravelli, A. Gurfinkel, and S. Chaki. SMT-based model checking for recursive programs. In *CAV*, pages 17–34, 2014.

[75] A. Komuravelli, A. Gurfinkel, and S. Chaki. Smt-based model checking for recursive programs. *Formal Methods in System Design*, 48(3):175–205, 2016.

[76] L. Lakhdar-Chaouch, B. Jeannet, and A. Girault. Widening with thresholds for programs with complex control graphs. In *ATVA*, pages 492–502, 2011.

[77] A. Lal and S. Qadeer. A program transformation for faster goal-directed search. In *FMCAD*, pages 147–154, 2014.

[78] C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88, 2004.

[79] C. Lattner and V. S. Adve. Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap. In *PLDI*, pages 129–142, 2005.

[80] K. McMillan and A. Rybalchenko. Solving Constrained Horn Clauses using Interpolation. Technical report, MSR-TR-2013-6, 2013.

[81] M. Méndez-Lojo, J. A. Navas, and M. V. Hermenegildo. A flexible, (C)LP-based approach to the analysis of object-oriented programs. In *LOPSTR*, pages 154–168, 2007.

[82] F. Merz, S. Falke, and C. Sinz. LLBMC: bounded model checking of C and C++ programs using a compiler IR. In *VSTTE*, pages 146–161, 2012.

[83] B. Mihaila, A. Sepp, and A. Simon. Widening as abstract domain. In *NASA NFM*, pages 170–184, 2013.

[84] A. Miné. A few graph-based relational numerical abstract domains. In *SAS*, pages 117–132, 2002.

[85] A. Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *LCTES*, pages 54–63, 2006.

[86] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[87] A. Miné. Symbolic methods to enhance the precision of numerical abstract domains. In *VMCAI*, pages 348–363, 2006.

[88] J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey. Signedness-agnostic program analysis: Precise integer bounds for low-level code. In R. Jhala and A. Igarashi, editors, *APLAS*, volume 7705 of *LNCS*, pages 115–130. Springer, 2012.

[89] C. Okasaki and A. Gill. Fast mergeable integer maps. In *Notes of the ACM SIGPLAN Workshop on ML*, pages 77–86, September 1998.

[90] J. C. Peralta, J. P. Gallagher, and H. Saglam. Analysis of imperative programs through analysis of constraint logic programs. In *SAS*, pages 246–261, 1998.

[91] Z. Rakamaric and M. Emmi. SMACK: Decoupling source language details from verifier implementations. In *CAV*, pages 106–113, 2014.

[92] X. Rival. Understanding the origin of alarms in astrée. In *SAS*, pages 303–319, 2005.

[93] P. Rümmer, H. Hojjat, and V. Kuncak. Disjunctive interpolants for horn-clause verification. In *CAV*, pages 347–363, 2013.

[94] A. Simon and A. King. Widening polyhedra with landmarks. In *APLAS*, pages 166–182, 2006.

[95] G. Singh, M. Püschel, and M. T. Vechev. ELINA: ETH Library for Numerical Analysis, 2018. Available at `https://github.com/eth-sri/ELINA`.

[96] N. Sinha, N. Singhania, S. Chandra, and M. Sridharan. Alternate and learn: Finding witnesses without looking all over. In *CAV*, pages 599–615, 2012.

[97] B. Steensgaard. Points-to analysis in almost linear time. In *POPL*, pages 32–41, 1996.

[98] TrailOfBits. Framework for lifting x86, amd64, and aarch64 program binaries to llvm bitcode. Available at `https://github.com/trailofbits/mcsema`.

[99] C. Urban, A. Gurfinkel, and T. Kahsai. Synthesizing ranking functions from bits and pieces. In *TACAS*, pages 54–70, 2016.

[100] A. Venet. A scalable nonuniform pointer analysis for embedded programs. In *SAS*, pages 149–164, 2004.

[101] A. Venet and G. P. Brat. Precise and efficient static array bound checking for large embedded C programs. In *PLDI*, pages 231–242, 2004.

[102] J. Zhao, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *POPL*, pages 427–440, 2012.